Design and Evaluation of a Code-Switching-Aware Multilingual Conversational AI System using Advanced RAG Architectures

A thesis submitted in partial fulfilment

of the requirements for the award of the degree

Master of Technology in Computer Science

by

Ashutosh Juvale

Roll No. CS2410

bridging languages, scripts & registers

Indian Statistical Institute, Kolkata

11 June 2026

Supervised by

Dr. Mendem Bapuji

Indian Statistical Institute, Kolkata

Dr. Niladri Sekhar Dash

Indian Statistical Institute, Kolkata

Certificate

Thesis title: *Code-Switching-Aware Multilingual RAG: Design and Evaluation of a Code-Switching-Aware Multilingual Conversational AI System using Advanced RAG Architectures*
Name of the student: Ashutosh Juvale **Roll No:** CS2410
Degree for which submitted: Master of Technology in Computer Science
Thesis supervisors: Dr. Mendem Bapuji and Dr. Niladri Sekhar Dash
Month and year of thesis submission: 11 June 2026

It is hereby certified that the work contained in this thesis titled **"Code-Switching-Aware Multilingual RAG"** by **Ashutosh Juvale** has been carried out under our supervision and that, to the best of our knowledge, it has not been submitted elsewhere for the award of any academic degree.

Dr. Mendem Bapuji
Indian Statistical Institute, Kolkata

Dr. Niladri Sekhar Dash
Indian Statistical Institute, Kolkata

Declaration

I hereby declare that the thesis titled **"Code-Switching-Aware Multilingual RAG"** has been authored by me. It presents the dissertation work conducted under the supervision of Dr. Mendem Bapuji and Dr. Niladri Sekhar Dash.

To the best of my knowledge, this is an original work in both research content and presentation. It has not been submitted elsewhere, in whole or in part, for any degree. All relevant prior work and collaborations have been duly acknowledged and cited in accordance with academic standards. The accompanying software has been released as open source.[1]

Ashutosh Juvale
Roll No. CS2410
Master of Technology in Computer Science
Indian Statistical Institute, Kolkata

[1] Source code: https://github.com/ashutoshjuvale/setu-rag

Abstract

Conversational artificial intelligence has become the primary interface through which hundreds of millions of users in India seek information and customer support. Yet the way these users actually write and speak is fundamentally at odds with the monolingual assumptions baked into most retrieval and generation systems: they *code-switch*, fluidly mixing one or more of the twenty-two scheduled languages of India with English, frequently typing Indic words in the Roman script ("*mera refund kab tak aayega*"). Standard Retrieval-Augmented Generation (RAG) pipelines silently fail on such input — the retriever returns off-topic passages because the query and the knowledge base live in different representation spaces, and the generator replies in a register that does not match the user.

This thesis presents SETU-RAG (*from sētu*, a bridge), a code-switching-aware multilingual RAG system engineered to run end-to-end on a single commodity GPU (a Google Colab T4 with 16 GB of memory). The system makes three novel contributions. First, a **CMI-Adaptive Retrieval Router** routes the retrieval strategy by the *linguistic* profile of the query — its Code-Mixing Index (CMI) and matrix language — rather than by reasoning complexity, so monolingual queries stay cheap while genuinely code-mixed queries trigger a crosslingual fan-out. Second, a **Transliteration-Robust Multi-View Query** expands every query into up to four parallel views (surface, native-script, matrix-canonical, and English-pivot), each embedded separately, so at least one view lands in the knowledge base's representation space regardless of script. Third, a **CMI-Conditioned Generation** stage conditions the answer on the user's measured matrix language and mix ratio so that the reply mirrors their register while remaining grounded in retrieved evidence. Around this text core we build a speech-to-speech (VANI) layer that adds two further contributions — acoustic–lexical language-identification fusion and CMI-conditioned text-to-speech — enabling code-switched voice in and style-matched voice out.

We additionally introduce **CS-RAGAS**, an evaluation harness that augments the standard RAGAS quality axes (faithfulness, answer relevancy, context precision/recall) with code-switching-native metrics: CMI-alignment, language-consistency, and transliteration-robustness. Every model in the pipeline is wired in a *real-with-fallback* manner — the live path loads strong open-weight models (BGE-M3, BGE-reranker-v2-m3, IndicLID, IndicXlit, IndicTrans2, and a 4-bit instruction-tuned generator), while a deterministic stand-in keeps the entire system runnable offline and on CPU for testing and reproducibility. We describe the design, the mathematical formulation of each stage, the corrective (CRAG) and faithfulness (Self-RAG) gates that guard against the

documented failure modes of code-switched retrieval, and the memory policy that keeps peak usage under 16 GB. Experiments on a code-switched customer-support corpus demonstrate that the router produces well-calibrated routing decisions, that the multi-view expansion recovers retrieval hits that a single-view retriever misses, and that the CS-native metrics capture register-mirroring behaviour that conventional metrics miss entirely. The work is a step toward conversational AI that meets Indian users in the language — and the script, and the register — in which they actually speak.

Source code. The complete implementation is publicly available at `https://github.com/ashutoshjuvale/setu-rag`.

Acknowledgements

I express my heartfelt gratitude to my supervisors, Dr. Mendem Bapuji and Dr. Niladri Sekhar Dash, whose unwavering support, insightful feedback, and constant encouragement have been invaluable throughout the course of this research. Their mentorship profoundly shaped both the direction and the depth of this work, and I feel truly fortunate to have learned under their guidance. Their willingness to let me chase an unconventional idea — that the *linguistics* of a query, not just its semantics, should steer a retrieval pipeline — gave this thesis its identity.

I am equally thankful to the faculty of the Computer Vision & Pattern Recognition Unit and the broader Indian Statistical Institute community, whose thoughtful advice greatly enriched my academic journey. I gratefully acknowledge the open-source ecosystem that made this work possible at all: the **AI4Bharat** group, whose IndicLID, IndicXlit, IndicTrans2, IndicConformer, and Indic Parler-TTS models are the backbone of the Indic-language front-end and speech layers; the **BAAI** team behind BGE-M3 and its reranker; and the maintainers of Hugging Face Transformers, sentence-transformers, FAISS, and `bitsandbytes`, without whose tooling a 16 GB GPU budget would have been impossible to meet.

My sincere thanks to my colleagues and seniors for being a constant source of inspiration and for the many conversations — often themselves cheerfully code-switched — that sharpened the ideas in this thesis. This work stands on the foundation of their collective wisdom and encouragement.

Finally, I thank my family for their patience and belief, and I dedicate the spirit of this work to the everyday multilingualism of the Indian subcontinent, which is not a problem to be solved so much as a richness to be honoured.

"The limits of my language mean the limits of my world."

— Ludwig Wittgenstein

Dedicated to every speaker who has ever switched mid-sentence,
and to the languages that flow into one another like rivers.

Contents

Abstract iii

Abbreviations x

Symbols xi

1 Introduction 1
- 1.1 The Multilingual, Multi-Script Reality of Indian Users 1
- 1.2 Why Code-Switching Breaks Standard RAG 2
- 1.3 Background: Retrieval-Augmented Generation in Brief 3
- 1.4 Problem Statement and Research Questions 3
- 1.5 Contributions 4
- 1.6 Design Constraints and Engineering Philosophy 4
- 1.7 Organisation of the Thesis 5

2 Background and Literature Review 7
- 2.1 Retrieval-Augmented Generation 7
- 2.2 Adaptive, Corrective, and Self-Reflective RAG 7
- 2.3 Multilingual and Cross-Lingual Retrieval 8
- 2.4 Code-Switching in Natural Language Processing 9
- 2.5 The Indic NLP and Speech Stack 9
- 2.6 Query Expansion and Rank Fusion 10
- 2.7 Evaluating RAG Systems 10
- 2.8 Positioning of This Thesis 11

3 System Overview and Problem Formulation 12
- 3.1 Design Goals 12
- 3.2 Formal Problem Statement 12
- 3.3 The Eight-Stage Pipeline 13
- 3.4 Knowledge-Base Data Model 15
- 3.5 The Real-with-Fallback Principle 15
- 3.6 Model Registry 16
- 3.7 T4 Memory Policy 16

4 Code-Switching-Aware Linguistic Front-End 19

4.1 Token-Level Language Identification 19
4.2 Script Detection 20
4.3 The Code-Mixing Index 20
4.4 Matrix Language, Embedded Languages, and Switch Points 22
4.5 A Worked Example 22
4.6 Transliteration Normalisation 22
4.7 Limitations of the Deterministic Fallback 23

5 CMI-Adaptive Retrieval Router **24**
5.1 Motivation: Route by Linguistics, Not by Reasoning 24
5.2 The Five Routes 25
5.3 Threshold Policy 25
5.4 The Decision Procedure 26
5.5 A Trainable Logistic Variant 27
5.6 Cost Analysis 27

6 Transliteration-Robust Multi-View Retrieval **28**
6.1 The Script-Mismatch Problem, Concretely 28
6.2 The Four Views 28
6.3 Embeddings: BGE-M3 30
6.4 Hybrid Index 31
6.5 Reciprocal Rank Fusion 31
6.6 Cross-Encoder Reranking 32
6.7 Corrective Grading (CRAG) 32
6.8 Summary 33

7 CMI-Conditioned Generation and Faithfulness **34**
7.1 The Register-Mismatch Problem 34
7.2 The CMI-Conditioned Style Directive 34
7.3 Prompt Construction and Grounding 35
7.4 Generator Loading under a 16 GB Budget 35
7.5 The Faithfulness Gate 36
7.6 A Worked Example of Register Mirroring 37
7.7 Summary 37

8 Speech-to-Speech Extension: The VANI Layer **39**
8.1 Design: A Wrapper, Not a Rewrite 39
8.2 Audio I/O and Voice-Activity Detection 39
8.3 Automatic Speech Recognition 40
8.4 Acoustic–Lexical LID Fusion 41
8.5 CMI-Conditioned Text-to-Speech 41

8.6 Turn Orchestration and Memory 42

9 Evaluation Methodology: CS-RAGAS **43**
9.1 Evaluation Task 43
9.2 Datasets 43
9.3 Conventional Quality Axes (RAGAS) 44
9.4 Code-Switching-Native Metrics 44
9.5 Speech Metrics 45
9.6 Experimental Setup 45

10 Results and Discussion **47**
10.1 Measured Offline Results 47
10.2 Route Distribution and CMI Calibration 48
10.3 Per-Query Traces 49
10.4 Retrieval Cost Model 49
10.5 Multi-View Ablation (Illustrative) 50
10.6 Qualitative Register Analysis 50
10.7 Projected Full-Model Quality (Illustrative) 51
10.8 Summary of Findings 52

11 Conclusion **53**
11.1 Summary 53
11.2 Contributions Revisited 53
11.3 Closing Remarks 54

12 Limitations and Future Work **55**
12.1 Limitations 55
12.2 Future Work 56

Bibliography **58**

List of Figures

1.1 The SETU-RAG architecture 6

2.1 A taxonomy of the relevant literature 8

3.1 The eight-stage pipeline as data flow 14

5.1 Router decision flow 25

6.1 Multi-view fan-out and fusion 30

8.1 The VANI speech-to-speech layer 40

10.1 Measured offline metrics 48
10.2 Measured route distribution 49
10.3 Retrieval cost by route 50
10.4 Measured vs. illustrative quality 51

List of Tables

3.1 Real-with-fallback components 16
3.2 Open-weight model registry 18

4.1 Unicode script ranges 20
4.2 Front-end features on real queries 22

5.1 The five retrieval routes 25

6.1 The four query views 29

7.1 Register mirroring across systems 38

9.1 Evaluation data 44

10.1 Measured offline metrics 48
10.2 Per-query traces 49
10.3 Expected effect of view ablation (illustrative) 51

Abbreviations

RAG	Retrieval-Augmented Generation
CRAG	Corrective Retrieval-Augmented Generation
CMI	Code-Mixing Index
CS	Code-Switching / Code-Switched
LID	Language Identification
LLM	Large Language Model
NLI	Natural Language Inference
IR	Information Retrieval
ConvQA	Conversational Question Answering
KB	Knowledge Base
RRF	Reciprocal Rank Fusion
BM25	Best Matching 25 (lexical ranking function)
ASR	Automatic Speech Recognition
TTS	Text-to-Speech
VAD	Voice Activity Detection
MT	Machine Translation
WER	Word Error Rate
CER	Character Error Rate
MOS	Mean Opinion Score
nDCG	Normalised Discounted Cumulative Gain
MRR	Mean Reciprocal Rank
NF4	4-bit NormalFloat quantization
VRAM	Video Random-Access Memory (GPU memory)
ML / EL	Matrix Language / Embedded Language
OOV	Out-of-Vocabulary
VANI	Voice-Aware Native Interface (the speech-to-speech layer)

Symbols

q	Query	The user's input question (text or transcribed speech)
$\mathcal{K}$	Knowledge base	The corpus of support documents / FAQ chunks
d	Document	A retrievable chunk in $\mathcal{K}$
CMI	Code-Mixing Index	Degree of code-mixing of an utterance, in $[0, 1]$
m	Matrix language	The dominant (grammatical-frame) language of q
E	Embedded language(s)	Non-matrix language(s) present in q
V	View set	The multi-view expansion $\{v_1, \ldots, v_n\}$ of q
ρ	Roman fraction	Fraction of language-bearing tokens in Latin script
σ	Switch points	Number of language transitions between adjacent tokens
RRF	Fusion function	Reciprocal Rank Fusion over ranked lists
k	RRF constant	Rank-discount constant in RRF (default 60)
K	Context size	Number of documents passed to the generator after rerank
S	Score function	Relevance / reranker score for a (query, document) pair
$\tau_{\mathrm{hi}}, \tau_{\mathrm{lo}}$	CMI thresholds	Router decision boundaries (`cmi_high`, `cmi_low`)
θ	Faithfulness gate	Minimum grounded-claim fraction required
α	Fusion weight	Weight on the acoustic prior in LID fusion

CHAPTER 1

Introduction

Large Language Models (LLMs) and the Retrieval-Augmented Generation (RAG) systems built on top of them have rapidly become the dominant interface through which people ask questions and obtain support. In India, this transition is happening at an extraordinary scale and against an extraordinary linguistic backdrop: the constitution recognises twenty-two scheduled languages, written in more than a dozen scripts, and the everyday digital language of hundreds of millions of users is not any one of them in isolation but a fluid blend of several of them with English. A user seeking a refund does not write "*When will my refund arrive?*" nor its clean Hindi equivalent in Devanagari; they write "*mera refund kab tak aayega, order cancel kar diya tha*" — Hindi grammar, English content words, the whole thing typed in the Roman script. This phenomenon, *code-switching*, is the rule rather than the exception, and it sits squarely in the blind spot of systems that were designed around a single language and a single script.

This thesis is about building a RAG system that treats code-switching not as noise to be cleaned away but as a first-class signal to be measured and acted upon. The result is SETU-RAG — from the Sanskrit *sētu*, meaning *a bridge* — a code-switching-aware multilingual conversational AI system whose retrieval, generation, and (optionally) speech stages are all conditioned on the linguistic profile of the user's utterance. fig. 1.1 shows the system end-to-end.

1.1 The Multilingual, Multi-Script Reality of Indian Users

The scale of Indian-language digital activity is no longer marginal. Hundreds of millions of users transact, search, and seek support online, and the majority do so in a language other than English — or, more precisely, in a *mixture* that defies the monolingual category altogether. Three properties of this usage matter for system design.

Code-switching is pervasive and structured. Speakers do not mix languages randomly; they switch at predictable syntactic boundaries, embedding content words from one language (often English technical or commercial vocabulary such as *refund*, *coupon*, *delivery*) inside the grammatical frame of another, the *matrix* language [34]. The degree of mixing varies continuously, from a single borrowed word to near-balanced alternation, and this degree is itself informative.

Romanisation is the default written form. Most Indian-language input on consumer keyboards is typed in the Roman script rather than in the language's native script, because Roman input is faster and universally available. The consequence for retrieval is severe: a knowledge base authored in native Devanagari, Tamil, or Bengali script shares almost no surface tokens with a romanised query, and dense embedders — though multilingual — are systematically weaker on romanised Indic text than on native-script text, because the latter is far better represented in their pre-training corpora.

Register expectations are reciprocal. A user who writes in a lightly code-mixed Hinglish register expects a reply in the same register. A grammatically perfect but tonally mismatched answer — pure formal Hindi in Devanagari, or pure English — reads as robotic and, in a customer-support setting, erodes trust.

1.2 Why Code-Switching Breaks Standard RAG

A conventional RAG pipeline has three moving parts: an *embedder* that maps the query and the documents into a shared vector space, a *retriever* that returns the nearest documents, and a *generator* that conditions an answer on those documents. Each part carries an implicit monolingual assumption, and each fails in a characteristic way under code-switching.

1. **Representation mismatch at retrieval.** When the query is romanised Hinglish and the knowledge base is native-script Hindi, the two occupy different regions of the embedding space. The retriever does not error; it silently returns plausible-looking but off-topic passages. This is the most dangerous failure mode precisely because it is invisible — there is no exception, only a wrong answer grounded in the wrong evidence.
2. **Lexical-channel collapse.** Sparse retrievers such as BM25 [31], which key on exact token overlap, lose their grip entirely when the query's tokens (romanised) never appear in the corpus (native script), discarding a signal that is otherwise complementary to dense retrieval.
3. **Register drift at generation.** Even with correct evidence, a general-purpose generator left unconditioned will answer in whatever register its decoding happens to favour, ignoring the user's matrix language and mix ratio.

The central observation

The failure mode that matters for Indian-language RAG is *linguistic*, not *logical*. The query is not hard to *reason* about; it is hard to *match* and hard to *mirror*. A system that detects *how* code-mixed a query is, and *which* language frames it, can spend its retrieval and generation budget exactly where the difficulty actually lies — and nowhere else.

This observation is what separates SETU-RAG from generic adaptive-RAG systems. Where prior adaptive approaches route by estimated *reasoning complexity* (single-hop vs. multi-hop) [18], SETU-RAG routes by a measured *code-mixing profile*, because that profile is what predicts retrieval breakage.

1.3 Background: Retrieval-Augmented Generation in Brief

RAG [23] augments a parametric language model with a non-parametric memory: instead of relying solely on what the model memorised during training, the system retrieves relevant passages from an external corpus at inference time and conditions generation on them. This decoupling has two attractive properties for customer support. First, the knowledge base can be updated without retraining the model, so a changed refund policy is reflected immediately. Second, answers can be *grounded* — traced to specific retrieved passages — which both improves factuality and enables verification. The price is that the system is only as good as its retrieval: an answer grounded in the wrong passage is confidently wrong. Modern refinements address this directly — *adaptive* routing decides *whether* and *how* to retrieve [18], *corrective* RAG grades the retrieved evidence and re-queries when it is weak [39], and *self-reflective* RAG critiques its own output for faithfulness [3]. SETU-RAG adopts all three ideas but re-purposes them around the code-switching signal.

1.4 Problem Statement and Research Questions

Informally, the problem this thesis addresses is the following.

> *Given a user query that may freely code-switch among the scheduled languages of India and English, possibly written in the Roman script, and a knowledge base authored predominantly in native scripts, produce a grounded, register-matched answer — accurately, and within a fixed and modest compute budget.*

We decompose this into four research questions that organise the technical chapters:

RQ1 (Routing).
Can the *linguistic* profile of a query — its Code-Mixing Index and matrix language — serve as a better basis for adapting the retrieval strategy than reasoning-complexity heuristics, and can it be computed cheaply enough to be worthwhile?

RQ2 (Retrieval robustness).
Can expanding a query into multiple script-normalised views recover retrieval hits that a single-view retriever misses on romanised, code-mixed input?

RQ3 (Register-matched generation).
Can conditioning the generator on the measured matrix language and mix ratio produce answers that mirror the user's register without sacrificing grounding?

RQ4 (Evaluation).
What additional metrics are needed to detect whether a RAG system *behaves* correctly under code-switching, beyond the conventional quality axes?

1.5 Contributions

This thesis makes the following contributions, realised as a single, runnable, open-source system.

Contribution 1 — CMI-Adaptive Retrieval Router

A router that selects the retrieval strategy by the query's Code-Mixing Index and matrix language rather than by reasoning complexity. Monolingual queries take a single cheap dense path; genuinely code-mixed queries trigger a crosslingual multi-view fan-out; chit-chat is short-circuited with no retrieval at all. The router is a transparent threshold policy with a drop-in trainable logistic variant (chapter 5).

Contribution 2 — Transliteration-Robust Multi-View Query

A query-expansion scheme that produces up to four parallel views — surface, native-script (via transliteration), matrix-canonical, and English-pivot (via translation) — each embedded separately and fused with Reciprocal Rank Fusion, so that at least one view matches the knowledge base regardless of the query's script. This directly targets the representation-mismatch failure of section 1.2 (chapter 6).

Contribution 3 — CMI-Conditioned Generation

A generation stage whose system prompt is synthesised from the user's measured matrix language and Code-Mixing Index, so the reply mirrors their register (Hinglish in $\rightarrow$ Hinglish out) while remaining grounded in the retrieved context, guarded by a Self-RAG-style faithfulness gate (chapter 7).

Beyond the text core, the speech-to-speech (VANI) layer contributes **acoustic–lexical LID fusion** (combining a spoken-language posterior with token-level LID over the transcript for a robust matrix-language decision) and **CMI-conditioned text-to-speech** (synthesising a Parler-TTS style description from the matrix language and mix ratio so the spoken reply mirrors the user), both detailed in chapter 8. Finally, we contribute **CS-RAGAS** (chapter 9), an evaluation harness that augments the standard RAGAS quality axes with code-switching-native metrics — *CMI-alignment*, *language-consistency*, and *transliteration-robustness*.

1.6 Design Constraints and Engineering Philosophy

Two constraints shaped every decision in this thesis.

A single commodity GPU. The entire system is engineered to run on a single Google Colab T4 with 16 GB of VRAM. This is a deliberate choice: it keeps the work reproducible by anyone with free-tier access, and it forces an honest accounting of cost. Models are loaded on demand and freed between stages, the generator runs in 4-bit precision [10], the machine translation models are distilled 200–320M variants, and FAISS stays on CPU. The memory policy is described in section 3.7.

Real-with-fallback. Every model wrapper in the system has two paths: a *live* path that loads a strong open-weight model when its dependencies and a GPU are available, and a deterministic *fallback* that keeps the whole pipeline running offline and on CPU. The transliterator falls back to an identity map, the embedder to a hashing embedding, the reranker to lexical overlap, the generator to extractive selection, and so on. This is not a collection of stubs to be filled in later — it is a design principle that makes the system testable end-to-end without downloads, and gracefully degradable when an optional component (for example, the AI4Bharat front-end or the speech models) is absent. section 3.5 formalises this philosophy.

1.7 Organisation of the Thesis

The remainder of this dissertation is organised as follows. Chapter 2 surveys the relevant literature across RAG, adaptive and corrective retrieval, multilingual and cross-lingual retrieval, code-switching in NLP, the Indic NLP stack, query expansion, rank fusion, and RAG evaluation, and positions SETU-RAG within it. Chapter 3 gives the formal problem statement, walks through the eight-stage pipeline, presents the knowledge-base data model, and states the real-with-fallback philosophy and T4 memory policy precisely. Chapter 4 describes the linguistic front-end — token-level language identification, script detection, the Code-Mixing Index, matrix-language estimation, and transliteration normalisation. Chapter 5, chapter 6, and chapter 7 present the three novel text contributions in turn: the CMI-adaptive router, the transliteration-robust multi-view retrieval stack (with hybrid retrieval, rank fusion, reranking, and the CRAG corrective loop), and CMI-conditioned generation (with the faithfulness gate). Chapter 8 presents the speech-to-speech VANI layer and its two contributions. Chapter 9 defines the CS-RAGAS evaluation methodology, the datasets, the model suite, and the experimental setup. Chapter 10 reports and discusses the experimental results. Chapter 11 concludes, and chapter 12 discusses limitations and future directions.

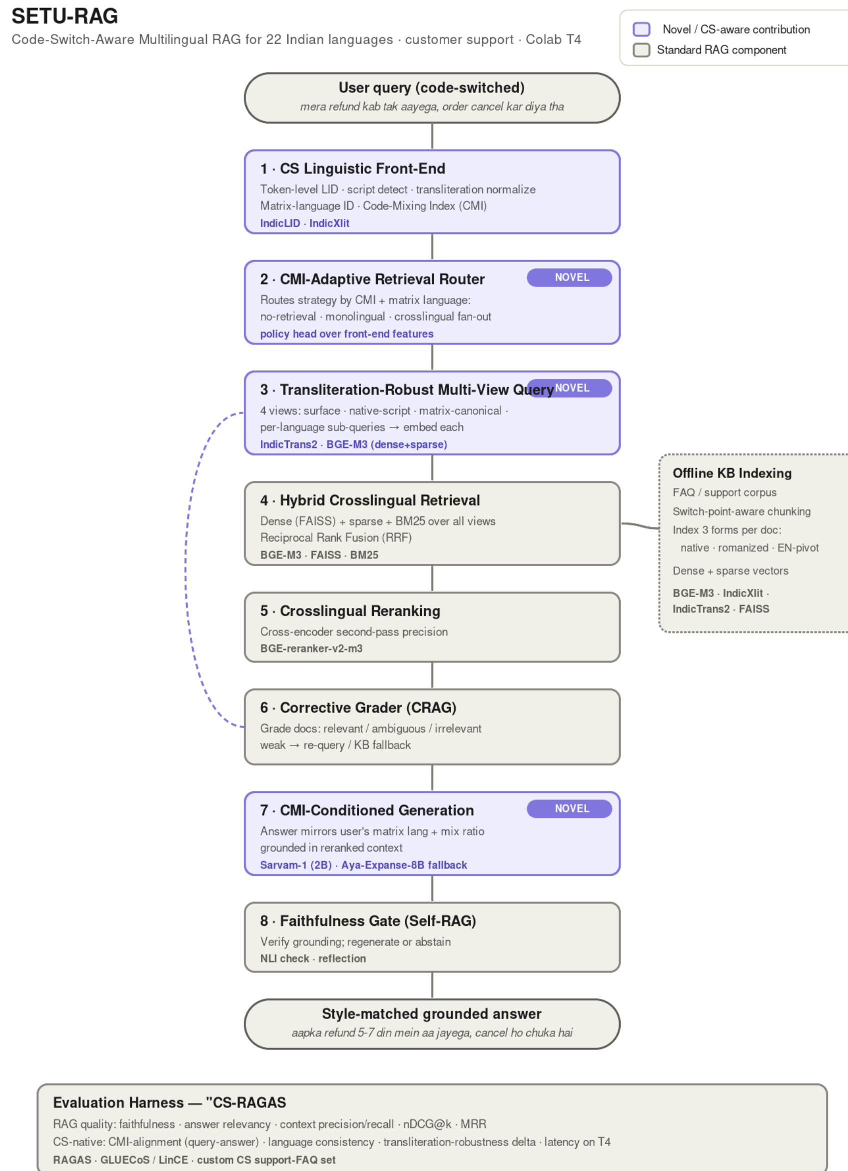

Figure 1.1. The SETU-RAG pipeline. A code-switched query flows through a linguistic front-end (token LID, script detection, transliteration normalisation, Code-Mixing Index and matrix-language estimation), a CMI-adaptive retrieval router, a transliteration-robust multi-view expansion, hybrid crosslingual retrieval with rank fusion and cross-encoder reranking, a corrective (CRAG) grader, CMI-conditioned generation, and a faithfulness gate. Purple blocks mark the novel, code-switching-aware contributions; grey blocks are otherwise-standard RAG components. The whole pipeline is engineered to run on a single 16 GB GPU.

CHAPTER 2

Background and Literature Review

This chapter surveys the bodies of work that SETU-RAG draws on and positions the system within them. The relevant literature spans seven threads: the foundations of retrieval-augmented generation; the family of adaptive, corrective, and self-reflective RAG variants; multilingual and cross-lingual retrieval; the computational treatment of code-switching; the open-source Indic NLP and speech stack; query expansion and rank fusion; and the evaluation of RAG systems. Figure 2.1 organises these threads and marks where this thesis contributes.

2.1 Retrieval-Augmented Generation

The modern RAG formulation is due to Lewis et al. [23], who coupled a dense passage retriever with a sequence-to-sequence generator and trained the two jointly, demonstrating gains on knowledge-intensive tasks while retaining the ability to update knowledge without retraining. The retrieval half of the system has its own rich lineage. *Dense* retrieval, exemplified by Dense Passage Retrieval [20], encodes queries and passages into a shared vector space with a dual encoder and retrieves by inner product; *late-interaction* methods such as ColBERT [22] preserve token-level granularity for finer matching; and sentence-embedding models in the SBERT lineage [30] made dense retrieval practical at scale. *Sparse* retrieval — classically BM25 [31] — remains a strong, interpretable baseline that keys on exact lexical overlap and is complementary to dense methods, which is precisely why *hybrid* retrieval that fuses the two is now standard. At billion-scale, approximate nearest-neighbour indices such as FAISS [19] make dense retrieval tractable. The embedding model used in this thesis, BGE-M3 [4], is notable for producing dense, sparse (learned-lexical), and multi-vector representations in a single forward pass over 100+ languages, which makes it an unusually good fit for a multilingual hybrid retriever.

2.2 Adaptive, Corrective, and Self-Reflective RAG

A naive RAG pipeline retrieves the same way for every query and trusts whatever it retrieves. Three lines of work relax these assumptions and are directly relevant to SETU-RAG.

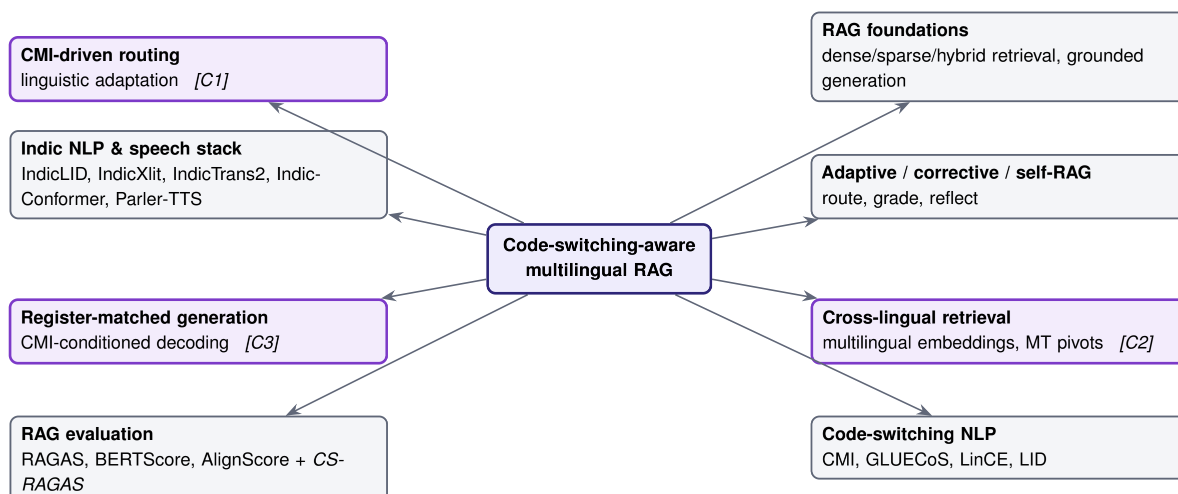

Figure 2.1. The literature threads that SETU-RAG synthesises. Purple leaves marked *[C1–C3]* indicate where the three novel contributions intervene; the evaluation leaf is extended by the CS-RAGAS harness.

Adaptive RAG. Jeong et al. [18] observe that queries vary in difficulty and that a one-size retrieval policy is wasteful: simple queries need no retrieval, single-hop queries need one retrieval, and complex queries need iterative retrieval. They train a classifier to predict query complexity and route accordingly. SETU-RAG inherits the *idea* of routing but replaces the routing *signal*: rather than estimating reasoning complexity, it measures a code-mixing profile, because in our setting the variation that predicts retrieval breakage is linguistic, not logical (chapter 5).

Corrective RAG (CRAG). Yan et al. [39] introduce a lightweight retrieval evaluator that grades retrieved documents as correct, ambiguous, or incorrect, and triggers corrective actions — decomposition, filtering, or a web search — when the evidence is weak. SETU-RAG adopts this grade-and-correct structure (section 6.7) but specialises the corrective action to a translation-based re-query, which is the action most likely to help when the failure is caused by a script or language mismatch rather than by genuine absence of the answer.

Self-reflective RAG (Self-RAG). Asai et al. [3] train a model to emit reflection tokens that decide when to retrieve and that critique whether the generated text is supported by the evidence, improving factuality. SETU-RAG adopts a lightweight, training-free analogue: a faithfulness gate that splits the answer into claims and checks each against the retrieved context, regenerating or abstaining when grounding is insufficient (section 7.5).

2.3 Multilingual and Cross-Lingual Retrieval

Multilingual encoders such as XLM-R [7] and multilingual E5 [37], and massively multilingual MT systems such as NLLB [27], made cross-lingual transfer the norm

rather than the exception. Yet cross-lingual *retrieval* — where the query and the corpus are in different languages or scripts — remains hard, and a recent survey of cross-lingual retrieval strategies for multilingual question answering [5] catalogues the main families: *translate the query* into the corpus language, *translate the corpus* into the query language, or use a *language-agnostic* embedding and retrieve directly. Each has failure modes — translation introduces errors, and language-agnostic embeddings degrade on under-represented scripts. SETU-RAG does not pick one strategy; its multi-view expansion (chapter 6) instantiates *several* of them in parallel — a native-script normalisation, an English pivot, and a matrix-canonical paraphrase — and lets rank fusion arbitrate, which is more robust than committing to any single channel. The romanisation problem in particular is what motivates the transliteration view: a romanised Indic query and a native-script corpus share neither surface tokens (defeating sparse retrieval) nor embedding neighbourhood (weakening dense retrieval), and normalising the script repairs both channels at once.

2.4 Code-Switching in Natural Language Processing

The computational study of code-switching is itself decades old [34, 38]. Two contributions from this literature are load-bearing for SETU-RAG. The first is the *Code-Mixing Index* (CMI) of Das and Gambäck [9], refined by Gambäck and Das [14], which quantifies how code-mixed an utterance is from the distribution of its per-token language tags and the number of switch points; SETU-RAG uses CMI as the central control signal for both routing and generation, and chapter 4 gives the exact formulation implemented. The second is the set of code-switching *benchmarks* — GLUECoS [21] and LinCE [1] — which established standard tasks (including language identification at the token level) and made clear that monolingual models degrade sharply on mixed input. Work on synthetic code-mixed data generation grounded in linguistic theory [28] further established that code-switching is rule- governed, which is the assumption underlying matrix-language estimation. What this literature has largely *not* done is fold the code-mixing signal back into a retrieval pipeline as a control variable; that is the gap this thesis targets.

2.5 The Indic NLP and Speech Stack

A practical Indian-language system lives or dies by the quality of its language-specific components, and the open-source ecosystem — much of it from the AI4Bharat group — has matured enough to build on. For *language identification*, IndicLID [24] is, to our knowledge, the only model that covers all twenty-two scheduled languages in *both* native and romanised script, which is exactly what token-level LID over romanised code-mixed text requires. For *transliteration*, IndicXlit, trained on the 26M-pair Aksharantar corpus [25], provides Roman↔ native conversion across 21 languages in a compact (∼11M-parameter) model. For *translation*, IndicTrans2 [13] offers high-quality MT across

all 22 languages, with distilled 200–320M checkpoints that fit a T4. On the *speech* side, IndicConformer (trained on the IndicVoices corpus [17], which deliberately includes code-switched speech) provides 22-language ASR and an acoustic LID head, Indic Parler-TTS [2] provides description-controllable multilingual synthesis, and Whisper [29] and Silero-VAD [33] serve as a general ASR fallback and a lightweight voice-activity detector respectively. The generators are Sarvam-1, an Indic-specialised 2B model [32], and Aya-Expanse-8B [6, 35] as a broad-coverage fallback. SETU-RAG's contribution is not any of these models individually but the way the code-mixing signal orchestrates them.

2.6 Query Expansion and Rank Fusion

Reformulating or expanding a query before retrieval is a long-standing way to bridge the vocabulary gap between queries and documents. Recent LLM-era methods include HyDE [15], which generates a hypothetical answer document and retrieves with *its* embedding, and Query2Doc [36], which prepends an LLM-generated pseudo-document to the query. SETU-RAG's multi-view expansion is in this spirit but is driven by *script and language* rather than by content: instead of imagining an answer, it re-expresses the *same* query in the representations most likely to match the corpus. When multiple expansions each produce a ranked list, those lists must be combined; Reciprocal Rank Fusion (RRF) [8] is the canonical, parameter-light way to do so, and it is what SETU-RAG uses to merge dense and sparse results across all views (section 6.5). RRF is attractive here precisely because it is rank-based and score-agnostic, so it fuses heterogeneous retrievers (a cosine-scored dense index and a BM25-scored sparse index) without score calibration.

2.7 Evaluating RAG Systems

Evaluating generated answers is harder than evaluating retrieval. Surface-overlap metrics such as BLEU and ROUGE penalise legitimate paraphrase; BERTScore [41] improves on them by comparing contextual embeddings, and AlignScore [40] measures factual consistency between a claim and its context with a model trained on millions of alignment examples. For RAG specifically, RAGAS [12] decomposes quality into faithfulness, answer relevancy, context precision, and context recall, computed with an LLM judge and without reference answers. The Massive Text Embedding Benchmark [26] provides the retrieval-side counterpart for embedding quality. None of these, however, asks whether a system behaves correctly *as a code-switching agent* — whether its answer mirrors the user's register, keeps their matrix language, and is robust to the script the query happens to be written in. chapter 9 introduces CS-RAGAS to fill exactly this gap, augmenting the RAGAS axes with CMI-alignment, language-consistency, and transliteration-robustness.

2.8 Positioning of This Thesis

In summary, the components SETU-RAG uses are individually well established: hybrid dense–sparse retrieval, RRF, cross-encoder reranking, CRAG-style correction, Self-RAG-style reflection, and the AI4Bharat Indic stack. The novelty is in the *organising principle*. Prior adaptive RAG routes by reasoning complexity; SETU-RAG routes by code-mixing profile. Prior query expansion bridges the vocabulary gap with content; SETU-RAG bridges the *script and language* gap with parallel views. Prior generation is left unconditioned on register; SETU-RAG conditions it on the measured matrix language and mix ratio. And prior evaluation measures generic quality; SETU-RAG adds code-switching-native behavioural metrics. The chapters that follow develop each of these in turn.

CHAPTER 3

System Overview and Problem Formulation

This chapter makes the problem precise, walks through the eight stages of the SETU-RAG pipeline at the level of data flowing between them, describes how the knowledge base is represented, and states the two engineering principles — real-with-fallback operation and a strict 16 GB memory budget — that constrain the whole design.

3.1 Design Goals

SETU-RAG is engineered against five goals, in rough priority order.

1. **Code-switch correctness.** Retrieval must find the right evidence and generation must mirror the user's register, even when the query is romanised and code-mixed and the corpus is native-script.
2. **Groundedness.** Every answer must be traceable to retrieved evidence; ungrounded answers must be caught and either regenerated or escalated to a human.
3. **Efficiency under a fixed budget.** The system must run end-to-end on a single 16 GB GPU, spending compute in proportion to linguistic difficulty.
4. **Modularity.** Each stage is a self-contained component with a clean interface, so individual models can be swapped without touching the rest.
5. **Reproducibility.** The system must run — if in a degraded mode — with no GPU and no model downloads, so that its structure and logic are testable by anyone.

3.2 Formal Problem Statement

We begin with the objects the system manipulates.

Definition 3.1 (Code-switched query)**.** A query is a sequence of tokens $q = (w_1, w_2, \ldots, w_T)$. Each token w_t carries a language tag $\ell_t \in \mathcal{L}$ and a script tag $s_t \in \mathcal{S}$, where $\mathcal{L}$ is the set of the twenty-two scheduled languages plus English and an "undetermined" class, and $\mathcal{S}$ is the set of Unicode scripts (Latin, Devanagari, Tamil, ...). A query is *code-switched* if its language-bearing tokens carry more than one distinct language family.

Definition 3.2 (Code-Mixing Index)**.** The Code-Mixing Index $\mathrm{CMI}(q) \in [0, 1]$ summarises the degree of code-mixing of q from the distribution of its per-token language

families and the number of switch points between adjacent tokens. CMI $= 0$ denotes a monolingual utterance and larger values denote heavier mixing. The exact form is given in section 4.3.

Definition 3.3 (Matrix and embedded languages)**.** The *matrix language* $m(q)$ is the most frequent language family among the language-bearing tokens of q; it is the grammatical frame of the utterance. The *embedded languages* $E(q)$ are the remaining families present.

Definition 3.4 (Knowledge base)**.** The knowledge base $\mathcal{K} = \{d_1, \ldots, d_M\}$ is a set of support documents (FAQ chunks). Each document d is authored in some language, predominantly in native script, and is indexed in multiple surface forms (section 3.4).

With these objects, the task is the following.

Problem 3.1 (Code-switch-aware grounded answering)**.** Given a query q and a knowledge base $\mathcal{K}$, produce an answer a that is (i) *grounded* — supported by a retrieved subset $C \subseteq \mathcal{K}$; (ii) *register-matched* — with $\mathrm{CMI}(a) \approx \mathrm{CMI}(q)$ and $m(a) = m(q)$; and (iii) *efficient* — produced within a fixed compute budget, with cost increasing in $\mathrm{CMI}(q)$ rather than being constant across queries.

The three clauses of this problem map onto the three novel contributions: efficiency-by-linguistic-difficulty is the router (chapter 5), grounded retrieval under script mismatch is multi-view retrieval (chapter 6), and register matching is CMI-conditioned generation (chapter 7).

3.3 The Eight-Stage Pipeline

Figure 3.1 renders the pipeline as a linear data-flow with the two feedback loops — the CRAG corrective re-query and the faithfulness regenerate — made explicit. We summarise each stage; later chapters elaborate.

Stage 1 — Front-end.
Token-level LID tags each word with a language and script; the Code-Mixing Index, matrix language, and the fraction of romanised tokens are computed from these tags (chapter 4).

Stage 2 — Router.
From $\mathrm{CMI}(q)$ and the romanised fraction, the router selects one of five routes (no-retrieval, mono-native, mono-roman, ambiguous, cross-lingual) and the number of query views to build (chapter 5).

Stage 3 — Multi-view query.
The chosen number of script-normalised views is constructed (chapter 6).

Stage 4 — Hybrid retrieval.
Each view is searched with both a dense index (BGE-M3 over FAISS) and a sparse index (BM25); the resulting ranked lists are fused with RRF (section 6.5).

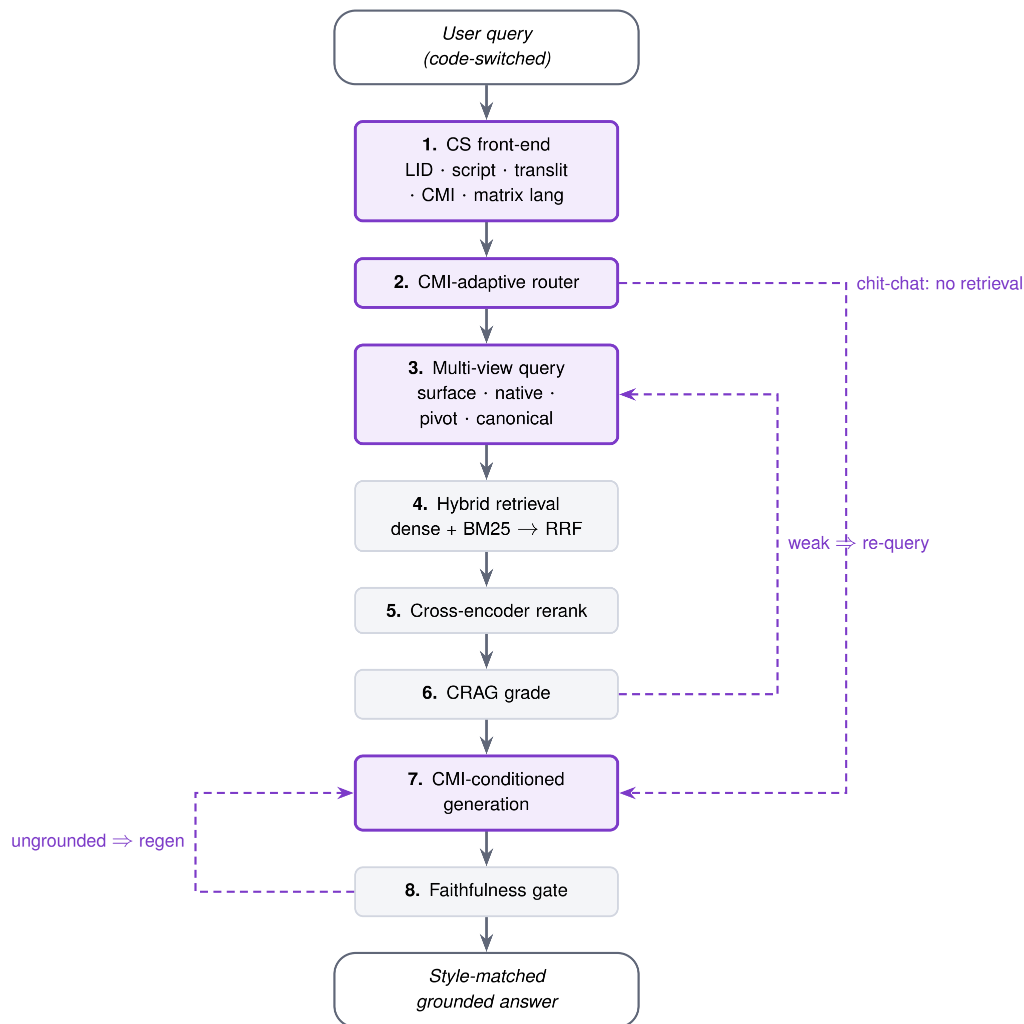

Figure 3.1. The pipeline as a data-flow graph. Dashed purple edges are control feedback: the router can bypass retrieval entirely for chit-chat, a weak CRAG grade triggers one corrective re-query, and an ungrounded answer triggers one regeneration before the faithfulness verdict is recorded.

Stage 5 — Reranking.
A cross-encoder (BGE-reranker-v2-m3) rescoring the fused candidates against the original query yields a precise top-k (section 6.6).

Stage 6 — CRAG grading.
The top rerank score and its margin grade the evidence as correct, ambiguous, or incorrect; weak evidence triggers a corrective English-pivot re-query (section 6.7).

Stage 7 — Generation.
The generator answers, conditioned on the retrieved context *and* on a style directive synthesised from $m(q)$ and $\mathrm{CMI}(q)$ (chapter 7).

Stage 8 — Faithfulness gate.
The answer is decomposed into claims and each is checked against the context; an ungrounded answer is regenerated or the system abstains and offers a human

handoff (section 7.5).

3.4 Knowledge-Base Data Model

The knowledge base is ingested offline. Crucially, each document is stored not as a single string but as a set of *surface forms*, so that whichever query view arrives, a matching form exists in the index. listing 3.1 shows the ingestion: each record keeps the original answer (native form), a romanised form (via transliteration), and an English-pivot form (via translation); when the transliterator or translator is absent, the forms gracefully collapse to the native text. A stable content hash serves as the document id, and switch-point-aware chunking keeps each chunk's matrix language coherent.

Listing 3.1. Offline KB ingestion: each chunk is stored in native, romanised, and English-pivot forms (`setu_rag/retrieval/kb_ingest.py`).

```
1  def expand_forms(chunk, lang, translit=None, translator=None):
2      roman = chunk
3      if translit is not None:
4          roman = " ".join(translit.native_to_roman(w, lang.split("_")[0])
5                           for w in chunk.split())
6      en_pivot = translator.to_english(chunk, lang) if translator else ""
7      return {"native": chunk, "roman": roman, "en_pivot": en_pivot}
8
9  def build_records(faq_path, translit=None, translator=None):
10     records = []
11     for row in read_jsonl(faq_path):
12         forms = expand_forms(row["answer"], row.get("lang", "hin_Deva"),
13                              translit, translator)
14         records.append({"id": chunk_id(row["answer"]), "lang": row.get("
               lang"),
15                         "question": row.get("question"), "answer": row["
                               answer"],
16                         "forms": forms, "meta": row.get("meta", {})})
17     return records
```

At index-build time (section 6.5) the searchable text for a document is the concatenation of its question, its native answer, and its romanised and English-pivot forms, so that a query in any of these representations can match on the lexical channel, while the dense channel embeds the same blob.

3.5 The Real-with-Fallback Principle

Every model-bearing component in SETU-RAG implements a common contract: a `load()` method that attempts the live model and, on any failure (missing dependency, no GPU, no network, or an explicit `force_offline` flag), silently installs a deterministic fallback and sets a `live` flag to `False`. table 3.1 lists the live path and fallback for each

Table 3.1. Each component's live path and its deterministic fallback. The fallbacks keep the whole pipeline runnable offline and on CPU.

Component	Live path	Deterministic fallback
Token LID	IndicLID	Unicode script + English-cue heuristic
Transliteration	IndicXlit	Identity pass-through
Translation	IndicTrans2 (distilled)	Empty string (view skipped)
Embedder	BGE-M3 (dense)	MD5 hashing n-gram embedding
Sparse retriever	BM25Okapi	Empty lexical list
ANN index	FAISS `IndexFlatIP`	NumPy brute-force cosine
Reranker	BGE-reranker-v2-m3	Jaccard lexical overlap
Generator	4-bit instruct LLM	Extractive (top passage)
Faithfulness	mDeBERTa NLI (optional)	Content-word grounding overlap
VAD	Silero-VAD	Whole clip as one segment
ASR	IndicConformer → Whisper	Empty transcript
Spoken-LID	IndicConformer LID head	Fixed (Hindi-biased) prior
TTS	Indic Parler-TTS	`None` (text-only reply)

component. The benefits are threefold: the entire pipeline is testable offline and on CPU with no downloads; optional enhancements (the AI4Bharat front-end, the speech models) can be installed incrementally without code changes; and a failure in any single component degrades that stage rather than crashing the system. The fallbacks are chosen to be *structurally faithful* — a hashing embedding still produces a vector space in which RRF and reranking are meaningful, even if its absolute quality is far below BGE-M3 — so that logic and plumbing can be validated independently of model quality.

3.6 Model Registry

table 3.2 lists the open-weight models the live pipeline loads, together with the rationale for each. All are selected to fit the T4 budget; the generator in particular defaults to an un-gated, multilingual, instruction-tuned model so the demonstration runs without a Hugging Face token, with Indic-specialised and broad-coverage alternatives available for the dissertation experiments.

3.7 T4 Memory Policy

The 16 GB budget is met by a strict load-on-demand, free-after-use policy. BGE-M3 (∼2.3 GB in fp16) and the reranker (∼2.3 GB) are loaded when first needed and released before the generator runs; FAISS stays on CPU so it never competes for VRAM; the translation models use the distilled 200–320M variants and are dropped between turns; and the generator runs in 4-bit NF4 precision (∼1.5–2 GB) [10, 11]. The generator's loader tries 4-bit, then fp16, then fp32, falling back along the way so that a machine

without bitsandbytes still runs (section 7.4). In the speech layer, the ASR and TTS models load sequentially within a turn and are freed between stages, so the ∼1.3 GB ASR and ∼1–2 GB TTS never coexist with the generator. The net effect is that, although the registry totals well over 16 GB if loaded at once, peak resident VRAM stays under the budget because no more than two of the heavy models are alive simultaneously.

Key runtime settings

The default configuration (Settings in config.py) uses retrieve_k= 30 candidates per view before fusion, rerank_k= 8 kept after the cross-encoder, max_ctx_docs= 5 passed to the generator, rrf_k= 60, CMI thresholds cmi_low= 0.05 and cmi_high= 0.20, and a faithfulness threshold of 0.7. These values recur throughout the technical chapters.

Table 3.2. The model registry (`setu_rag/config.py`). Every model is open-weight and selected to fit a single 16 GB T4.

Stage	Model	Why
Token LID	`ai4bharat/IndicLID`	Only LID covering all 22 langs incl. romanised
Transliteration	`ai4bharat/indicxlit`	Roman↔native, 21 langs, ∼11M params
Translation (3 dir.)	`ai4bharat/indictrans2-*-dist-200/320M`	22-lang MT, distilled to fit T4
Embedder	`BAAI/bge-m3`	Dense+sparse+ColBERT in one pass, 100+ langs, MIT
Reranker	`BAAI/bge-reranker-v2-m3`	Multilingual cross-encoder
Generator (demo)	`Qwen/Qwen2.5-3B-Instruct`	Un-gated, ∼2 GB in 4-bit, good Hindi/Indic
Generator (primary)	`sarvamai/sarvam-1`	Indic-specialised 2B, efficient tokenizer
Generator (fallback)	`CohereForAI/aya-expanse-8b`	23 langs, 4-bit on T4
Faithfulness NLI	`MoritzLaurer/mDeBERTa-v3-...-xnli`	Small multilingual entailment judge
ASR	`ai4bharat/indic-conformer-600m`	22 langs, code-switched (IndicVoices)
ASR fallback	`openai/whisper-large-v3-turbo`	General baseline, word timestamps
TTS	`ai4bharat/indic-parler-tts`	21 langs, code-mixing, description-controllable
VAD	`snakers4/silero-vad`	Tiny, CPU

CHAPTER 4

Code-Switching-Aware Linguistic Front-End

Everything downstream in SETU-RAG — the routing decision, the choice of query views, the generation register — is driven by a small set of linguistic features computed once, at the front of the pipeline, from the raw query. This chapter describes how those features are produced: token-level language identification (section 4.1), script detection (section 4.2), the Code-Mixing Index (section 4.3), matrix-language and switch-point estimation (section 4.4), and transliteration normalisation (section 4.6). A worked example (section 4.5) shows the features the front-end produces on real queries, and section 4.7 is candid about the limitations of the deterministic fallback.

4.1 Token-Level Language Identification

Code-mixing is, by definition, a per-token phenomenon: within one utterance, different words belong to different languages. The front-end therefore performs language identification at the *token* level rather than labelling the whole utterance with a single language. Each token is represented by the record in listing 4.1: its surface text, a language tag in the FLORES-style `xxx_Yyyy` format (e.g. `hin_Deva` for native Hindi, `hin_Latn` for romanised Hindi, `eng_Latn` for English), its script, and a confidence.

Listing 4.1. The token record produced by the LID stage (`setu_rag/front_end/language_id.py`).

```
@dataclass
class Token:
    text: str
    lang: str    # e.g. "hin_Deva", "eng_Latn", "hin_Latn" (romanized
        Hindi)
    script: str  # "Latn" | "Deva" | "Beng" | ...
    conf: float
```

The live path uses IndicLID [24], which — uniquely among available LID models — covers all twenty-two scheduled languages in *both* native and romanised script. This is exactly the capability token-level LID over romanised code-mixed text requires: the hard discrimination is between English and romanised Indic (is *"order"* English, or a

Table 4.1. The Unicode block ranges used for script detection. A token is assigned the script of its first in-range character; tokens with no in-range character are Latin.

Script	Range	Script	Range
Devanagari (Deva)	`U+0900–097F`	Gujarati (Gujr)	`U+0A80–0AFF`
Bengali (Beng)	`U+0980–09FF`	Malayalam (Mlym)	`U+0D00–0D7F`
Tamil (Taml)	`U+0B80–0BFF`	Kannada (Knda)	`U+0C80–0CFF`
Telugu (Telu)	`U+0C00–0C7F`	Oriya (Orya)	`U+0B00–0B7F`
Gurmukhi (Guru)	`U+0A00–0A7F`	Latin (Latn)	default

romanised Indic word?), and a model that has seen romanised Indic in training makes this distinction far more reliably than any script-based heuristic could. Words are batched to the model; punctuation and numbers are tagged locally as "undetermined" so they are excluded from the code-mixing computation and do not waste model calls. When IndicLID is unavailable, the front-end falls back to the deterministic heuristic of section 4.7.

4.2 Script Detection

Independent of the language model, each token's script is determined directly from the Unicode code points of its characters. This is fast, exact, and never wrong for native scripts, and it serves two purposes: it provides the script tag in the `Token` record, and it gives the fallback LID heuristic a strong prior (a token in the Tamil block is Tamil). table 4.1 lists the ranges used; any token whose characters fall outside these blocks is classified as Latin (Roman) script.

4.3 The Code-Mixing Index

The single most important feature the front-end produces is the Code-Mixing Index. We follow the formulation of Das and Gambäck [9] and Gambäck and Das [14], combining two intuitions: an utterance is more code-mixed (i) the more evenly its tokens are distributed across languages, and (ii) the more often it switches between languages as it proceeds. Let q have N language-bearing tokens (punctuation and numbers excluded), drawn from a set of language families with counts $\{w_1, \ldots, w_c\}$, let $\max_i w_i$ be the count of the most frequent family, and let σ be the number of switch points — adjacent token pairs whose families differ. The implemented index is

$$\mathrm{CMI}(q) = \underbrace{\tfrac{1}{2}\left(1 - \tfrac{\max_i w_i}{N}\right)}_{\text{fraction-embedded term}} + \underbrace{\tfrac{1}{2} \cdot \tfrac{\sigma}{N-1} \cdot \mathbb{1}[\, c > 1\,]}_{\text{switch-point term}}, \qquad N > 1, \tag{4.1}$$

and $\mathrm{CMI}(q) = 0$ when $N \leq 1$. The first term is the fraction of tokens *not* in the dominant language — it is zero for a monolingual utterance and approaches $\frac{1}{2}$ as the languages

Algorithm 1: ComputeCMI — Code-Mixing Index and matrix language

Input : Token sequence $(t_1, \ldots, t_T)$ with language tags
Output: $(\mathrm{CMI}, m, E, \sigma, N)$

1. $L \leftarrow [\,\mathrm{family}(t.\mathrm{lang}) : t \in \mathrm{tokens},\ t$ alphanumeric and language-bearing $]$;
2. $N \leftarrow |L|$;
3. **if** $N = 0$ **then**
4.   **return** (0, "und", $\emptyset$, 0, 0);
5. counts $\leftarrow$ Counter(L); $m \leftarrow \arg\max_\ell \mathrm{counts}[\ell]$; $\max_w \leftarrow \mathrm{counts}[m]$;
6. $\sigma \leftarrow \big|\{\, i : L_i \neq L_{i+1} \,\}\big|$;
7. frac $\leftarrow 1 - \max_w / N$;
8. sp $\leftarrow \sigma/(N-1)$ **if** $N > 1$ **else** 0;
9. CMI $\leftarrow 0.5 \cdot \mathrm{frac} + 0.5 \cdot \mathrm{sp} \cdot \mathbb{1}[\,|\mathrm{counts}| > 1\,]$;
10. $E \leftarrow \{\ell \in \mathrm{counts} : \ell \neq m\}$;
11. **return** (CMI, m, E, σ, N);

balance. The second term normalises the switch count by the maximum possible number of switches $N-1$, gated by an indicator that more than one language is actually present (so a monolingual utterance contributes nothing even if a single token is mis-tagged). The two terms are weighted equally, yielding CMI $\in [0, 1]$. algorithm 1 states the computation, and listing 4.2 is the implementation.

Listing 4.2. The CMI computation (`setu_rag/front_end/cmi.py`); romanised and native forms of the same language collapse to one family, so `hin_Latn` and `hin_Deva` both count as `hin`.

```python
def compute_cmi(tokens):
    langs = [lang_family(t.lang) for t in tokens
             if t.lang not in NON_LANG and t.text.isalnum()]
    n = len(langs)
    if n == 0:
        return CMIResult(0.0, "und", [], 0, 0)
    counts = Counter(langs)
    matrix, max_w = counts.most_common(1)[0]
    sp = sum(1 for a, b in zip(langs, langs[1:]) if a != b)
    frac_embedded = 1.0 - (max_w / n)
    sp_term = (sp / (n - 1)) if n > 1 else 0.0
    cmi = 0.5 * frac_embedded + 0.5 * sp_term * (1 if len(counts) > 1 else
        0)
    embedded = [l for l in counts if l != matrix]
    return CMIResult(round(cmi, 4), matrix, embedded, sp, n)
```

Why romanised and native forms collapse to one family

A token tagged `hin_Latn` (romanised Hindi) and one tagged `hin_Deva` (native Hindi) are the *same* language differing only in script. For the purpose of measuring

Table 4.2. Front-end output on four queries (measured, offline configuration). m is the matrix language, E the embedded languages, σ the switch-point count, and N the number of language-bearing tokens. The route column previews the router of chapter 5.

Query	CMI	m	E	σ	N	**Route**
mera refund kab tak aayega, maine order cancel kiya tha	0.37	hin	eng	4	10	cross-lingual
how do I track my order	0.28	eng	hin	2	6	cross-lingual
coupon apply nahi ho raha hai	0.18	hin	eng	1	6	ambiguous
order track karne ke liye Orders page par jaayein	0.35	hin	eng	3	9	cross-lingual

code-*mixing* we collapse both to the family `hin` (via `lang.split("_")[0]`), so that an all-Hindi sentence typed in Roman script correctly registers as monolingual (CMI $\approx$ 0) rather than as a Hindi/English mix. Script differences are handled separately, by the transliteration view (chapter 6), not by the code-mixing measure.

4.4 Matrix Language, Embedded Languages, and Switch Points

The same pass that computes CMI yields three features that the rest of the pipeline consumes. The *matrix language* m is the most frequent family — the grammatical frame, and the language in which the answer should be written. The *embedded languages* E are the remaining families; they determine which additional target languages a crosslingual fan-out should consider. The *switch-point count* σ is a structural signal of mixing intensity that feeds eq. (4.1). Together with the *romanised fraction* ρ — the fraction of language-bearing tokens in Latin script, computed directly from the script tags — these constitute the entire feature set the router (chapter 5) needs.

4.5 A Worked Example

table 4.2 shows the features the front-end actually produces (running the implementation in its offline configuration) on four representative queries. The romanised Hinglish refund query is correctly identified as Hindi-matrix with English embedded and a high CMI; the lightly-mixed coupon query lands in the ambiguous band; and the matrix language flips to English on the English-heavy tracking query, exactly as the downstream router requires.

4.6 Transliteration Normalisation

Because romanisation is the default written form (section 1.1) but knowledge bases and embedders are strongest in native script, the front-end provides a transliteration step that rewrites romanised Indic tokens into their native script. The live path uses IndicXlit [25]; the `normalize_query` method (listing 4.3) walks the tokens and transliterates each

romanised Indic token — using its own LID-detected language when available — while leaving English tokens and already-native tokens untouched. This single step is what makes the native-script query view of chapter 6 possible, and it repairs both the dense and sparse retrieval channels at once. When IndicXlit is absent, every transliteration method is an identity pass-through, and the native view simply collapses to the surface view.

Listing 4.3. Transliteration normalisation: romanised Indic tokens are rewritten to native script using their detected language (`setu_rag/front_end/transliterate.py`).

```
1  def normalize_query(self, tokens, default_lang="hi"):
2      out = []
3      for t in tokens:
4          if (t.script == "Latn"
5                  and t.lang.startswith(("hin","ben","tam","tel","mar","guj"
                       ,
6                                         "kan","mal","ory","pan","und"))
7                  and t.lang != "eng_Latn"):
8              lang = default_lang if t.lang.startswith("und") else t.lang
9              out.append(self.roman_to_native(t.text, lang))
10         else:
11             out.append(t.text)
12     return " ".join(out)
```

4.7 Limitations of the Deterministic Fallback

When IndicLID is unavailable, the front-end falls back to a script-plus-cue heuristic: native-script tokens are tagged by their script, Latin tokens that appear in a small English cue list are tagged English, and all other Latin alphabetic tokens are assumed to be romanised Hindi. This keeps the pipeline running, but it is deliberately crude, and being explicit about its failure modes matters for interpreting the offline results in chapter 10. For instance, on *"how do I track my order"* the function word *"do"* is not in the cue list and is therefore mis-tagged as romanised Hindi, which inflates the measured CMI and flips the matrix language; the live IndicLID path, having been trained on romanised Indic, does not make this error. The heuristic also cannot distinguish *which* Indic language a romanised token belongs to, defaulting to Hindi. These are limitations of the *fallback*, not of the design: the live front-end uses IndicLID precisely because it resolves exactly these ambiguities. We report offline numbers throughout as a demonstration that the *plumbing* is correct and the metrics are well-defined, and we are careful to label them as the deterministic-fallback configuration.

CHAPTER 5

CMI-Adaptive Retrieval Router

This chapter presents the first novel contribution: a retrieval router that decides *how* to retrieve based on the linguistic profile of the query. We motivate routing by code-mixing rather than by reasoning complexity (section 5.1), define the five routes and the threshold policy that selects among them (sections 5.2 and 5.3), give the decision procedure (section 5.4), describe a drop-in trainable variant (section 5.5), and analyse the cost the router saves (section 5.6).

Contribution 1 in one sentence

Spend retrieval effort in proportion to how code-mixed the query is: send genuinely monolingual queries down a single cheap dense path, fan a genuinely code-mixed query out across multiple script-normalised views, and short-circuit chit-chat with no retrieval at all.

5.1 Motivation: Route by Linguistics, Not by Reasoning

Adaptive-RAG [18] established that routing the retrieval strategy by query difficulty saves substantial compute, and it estimated difficulty as *reasoning complexity* — whether a question is non-retrieval, single-hop, or multi-hop. That signal is the right one for open-domain multi-hop QA. It is the *wrong* signal for our setting. In code-switched customer support, the questions are overwhelmingly single-hop and factual ("when does my refund arrive?"); what varies dramatically from query to query is not how much *reasoning* they require but how much *linguistic* work retrieval will need to find the answer. A clean monolingual native-script query matches the corpus directly and needs nothing fancy; a heavily romanised, code-mixed query needs script normalisation and crosslingual expansion or it will silently retrieve the wrong passage (section 1.2).

SETU-RAG's router therefore replaces the difficulty signal with the Code-Mixing Index and the matrix language from the front-end. This is both more appropriate to the failure mode *and* far cheaper to compute — the routing features are a by-product of the LID pass that has already run, so the router adds essentially no latency of its own, in contrast to a learned complexity classifier that requires its own forward pass.

Table 5.1. The routes the CMI-adaptive router selects among. ρ is the romanised fraction; τ_{lo} and τ_{hi} are the CMI thresholds (`cmi_low`$= 0.05$, `cmi_high`$= 0.20$).

Route	Condition	Views	Rationale
NO_RETRIEVAL	chit-chat / greeting	0	nothing to ground; answer directly
MONO_NATIVE	CMI $< \tau_{lo}$, $\rho \leq 0.5$	1	native-script monolingual; single dense query suffices
MONO_ROMAN	CMI $< \tau_{lo}$, $\rho > 0.5$	2	romanised monolingual; add a transliterated native view
AMBIGUOUS	$\tau_{lo} \leq$ CMI $< \tau_{hi}$	2	lightly mixed; a light fan-out hedges the script
CROSS_LINGUAL	CMI $\geq \tau_{hi}$	4	genuinely code-mixed; full multi-view fan-out

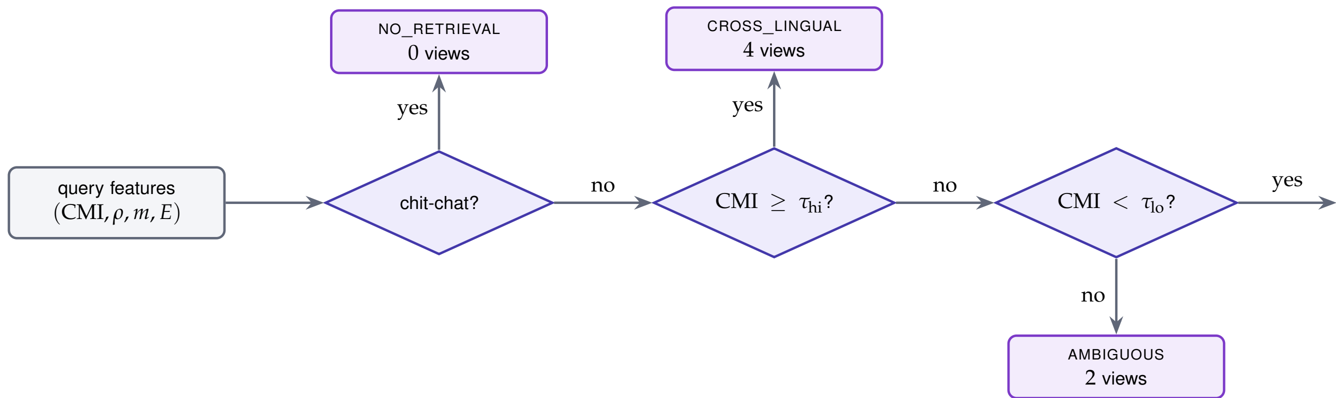

Figure 5.1. The router decision flow. Chit-chat is short-circuited; otherwise the Code-Mixing Index selects between a full fan-out, a light fan-out, and a monolingual path, with the romanised fraction ρ deciding whether a monolingual query still needs a transliterated view.

5.2 The Five Routes

The router maps the front-end features to one of five routes, each prescribing a retrieval strategy and a number of query views to build. table 5.1 defines them.

Figure 5.1 renders the decision as a flow. The key property is monotonicity: as CMI rises, the route never prescribes *fewer* views, so retrieval cost increases smoothly with linguistic difficulty.

5.3 Threshold Policy

The two thresholds carve the CMI axis into three bands. Below $\tau_{lo} = 0.05$ a query is treated as *effectively monolingual* — a single stray token does not justify a crosslingual fan-out. Above $\tau_{hi} = 0.20$ a query is treated as *genuinely code-mixed* and gets the full four-view expansion. Between the thresholds lies an *ambiguous* band that receives a light two-view hedge. The values 0.05 and 0.20 were chosen so that a sentence with one borrowed content word out of ten or more tokens stays monolingual, while a sentence

Algorithm 2: DECIDEROUTE — CMI-adaptive routing

Input : CMI result (CMI, m, E); romanised fraction ρ; chit-chat flag z
Output: Route, number of views, target languages

1 **if** z **then**
2   **return** (NO_RETRIEVAL, 0, $\emptyset$);
3 **if** $\mathrm{CMI} \geq \tau_{\mathrm{hi}}$ **then**
4   **return** (CROSS_LINGUAL, 4, $[m] \parallel E$);
5 **if** $\mathrm{CMI} < \tau_{\mathrm{lo}}$ **then**
6   **if** $\rho > 0.5$ **then return** (MONO_ROMAN, 2, $[m]$);
7   **return** (MONO_NATIVE, 1, $[m]$);
8 **return** (AMBIGUOUS, 2, $([m] \parallel E)[:2]$);

with two or more switches crosses into the code-mixed regime; section 4.5 shows real queries landing in each band. Within the monolingual band, the romanised fraction ρ further distinguishes a native-script query (which matches the corpus directly, one view) from a romanised query (which needs a transliterated view to match, hence two).

5.4 The Decision Procedure

algorithm 2 states the procedure and listing 5.1 is the implementation: a handful of comparisons over already-computed features. For the cross-lingual route, the target language list is the matrix language followed by its embedded languages, which the multi-view builder (chapter 6) uses to decide which pivots to construct.

Listing 5.1. The router (`setu_rag/router/adaptive_router.py`).

```
1  def decide_route(cmi, frac_roman, chit_chat=False):
2      s = SETTINGS
3      if chit_chat:
4          return RouteDecision(Route.NO_RETRIEVAL, 0, [])
5      if cmi.cmi >= s.cmi_high:
6          langs = [cmi.matrix_lang, *cmi.embedded_langs] or [cmi.matrix_lang
               ]
7          return RouteDecision(Route.CROSS_LINGUAL, 4, langs)
8      if cmi.cmi < s.cmi_low:
9          if frac_roman > 0.5:
10             return RouteDecision(Route.MONO_ROMAN, 2, [cmi.matrix_lang])
11         return RouteDecision(Route.MONO_NATIVE, 1, [cmi.matrix_lang])
12     return RouteDecision(Route.AMBIGUOUS, 2,
13                          [cmi.matrix_lang, *cmi.embedded_langs][:2])
```

5.5 A Trainable Logistic Variant

The threshold policy is transparent, requires no training data, and is easy to audit — valuable properties for a deployed support system. When labelled routing data is available, however, the same interface admits a learned variant that replaces the hand-set thresholds with a small logistic head over the feature vector

$$\boldsymbol{x}(q) \;=\; \left[\mathrm{CMI}(q),\, \tfrac{\sigma}{N},\, \rho,\, |E|,\, \mathbb{1}[\,m = \mathrm{eng}\,]\right]^{\top}, \tag{5.1}$$

i.e. the Code-Mixing Index, the normalised switch-rate, the romanised fraction, the number of embedded languages, and an indicator of whether the matrix language is English. A multinomial logistic classifier $\mathrm{softmax}(W\boldsymbol{x} + \boldsymbol{b})$ over the four retrieval routes can be fit to maximise downstream retrieval hit-rate per unit cost, learning the band boundaries from data rather than fixing them at 0.05 and 0.20. Because the feature vector is tiny and computed from existing front-end output, the learned router remains essentially free at inference. We use the threshold policy in this thesis and leave the learned head as a documented extension, since the sample corpus is too small to fit it without overfitting.

5.6 Cost Analysis

The value of routing is the retrieval work it avoids on easy queries. Let the per-view retrieval cost be c_{view} (dominated by embedding the view and the two index searches it triggers). A non-adaptive baseline that always builds the full four-view fan-out pays $4\,c_{\text{view}}$ on *every* query. The adaptive router pays $n(q)\,c_{\text{view}}$ where $n(q) \in \{0, 1, 2, 4\}$ depends on the route. For a workload with route mix $(\pi_{\text{NR}}, \pi_{\text{MN}}, \pi_{\text{MR}}, \pi_{\text{AMB}}, \pi_{\text{CL}})$, the expected per-query view count is

$$\mathbb{E}[n] \;=\; 0 \cdot \pi_{\text{NR}} + 1 \cdot \pi_{\text{MN}} + 2\,(\pi_{\text{MR}} + \pi_{\text{AMB}}) + 4\,\pi_{\text{CL}}, \tag{5.2}$$

and the speed-up over the always-fan-out baseline is $4/\mathbb{E}[n]$. A workload skewed toward monolingual and lightly-mixed queries — the common case in practice — drives $\mathbb{E}[n]$ well below 4 and recovers a multiplicative saving on the single most expensive part of the pipeline. Section 10.4 instantiates eq. (5.2) with the measured route distribution. Crucially, the router buys this saving without sacrificing quality on the hard queries, because those are exactly the ones it still fans out fully.

CHAPTER 6

Transliteration-Robust Multi-View Retrieval

This chapter presents the second novel contribution — the transliteration-robust multi-view query — together with the full retrieval stack it feeds: hybrid dense–sparse retrieval, Reciprocal Rank Fusion, cross-encoder reranking, and the CRAG corrective loop. The multi-view expansion (section 6.2) is the heart of the contribution; the remaining sections describe how the views are searched (sections 6.3 to 6.5), refined (section 6.6), and guarded (section 6.7).

Contribution 2 in one sentence

Re-express the same query in up to four representations — as typed, transliterated to native script, paraphrased into clean matrix-language script, and pivoted to English — embed each separately, and fuse the results, so at least one view lands in the knowledge base's representation space whatever script the query was written in.

6.1 The Script-Mismatch Problem, Concretely

Consider the romanised Hinglish query *"mera refund kab tak aayega"* against a knowledge base whose refund FAQ is stored as a native-script Hindi answer that means *"a refund reaches your original payment method in 5–7 business days"*. The query and the document share the English content words *refund*, *business*, *days*, *payment* — but the Hindi function words that carry the grammatical match are in different scripts in the general case, and a dense embedder trained mostly on native-script Hindi places the romanised query and the native document in different neighbourhoods. The result is a retrieval that is plausible but wrong. The multi-view expansion repairs this by ensuring that, for any query, there exists a view whose representation matches the corpus: a native-script view to match native documents on both the dense and the lexical channels, and an English-pivot view to match on the English content that survives code-switching.

6.2 The Four Views

Given a query q, its token tags, its CMI result, and the router's decision, the multi-view builder constructs an ordered set of views $V(q) = \{v_1, \ldots, v_n\}$, $n \leq 4$, deduplicated by normalised text. table 6.1 defines them.

Table 6.1. The query views and the route at which each is activated. The transliterator and translator are optional; when absent, the views that depend on them are skipped and the system proceeds on the remaining views.

View	Built when	What it is
SURFACE	always	the query exactly as typed; preserves the user's own tokens
NATIVE	mono-roman, ambiguous, cross-lingual	romanised Indic tokens transliterated to native script (IndicXlit)
ENGLISH_PIVOT	ambiguous, cross-lingual	the query translated to English (IndicTrans2); matches English KB content
MATRIX_CANON	cross-lingual	a clean native-script paraphrase of the query in its matrix language (pivoted through English)

Algorithm 3: BUILDVIEWS — transliteration-robust multi-view expansion

Input : Query q; tokens; CMI result; route decision r
Output: Deduplicated view set V

1 $V \leftarrow [\,(\text{SURFACE}, q, m)\,]$;
2 **if** $r.route \in \{\text{MONO_ROMAN}, \text{AMBIGUOUS}, \text{CROSS_LINGUAL}\}$ ***and*** *translit available* **then**
3     $u \leftarrow \text{NORMALIZEQUERY}(\text{tokens})$;
4   **if** $u \neq q$ ***and*** u *nonempty* **then** append (NATIVE, u, m) to V;
5 **if** $r.route \in \{\text{AMBIGUOUS}, \text{CROSS_LINGUAL}\}$ ***and*** *translator available* **then**
6   append $(\text{ENGLISH_PIVOT}, \text{TOENGLISH}(q, m), \text{eng})$ to V;
7 **if** $r.route = \text{CROSS_LINGUAL}$ ***and*** *translator available* **then**
8   append $(\text{MATRIX_CANON}, \text{TOLANG}(q, m, m), m)$ to V;
9 **return** dedup(V) by lowercased text;

The construction is given in algorithm 3 and implemented in listing 6.1. Two design points are worth highlighting. First, the views are gated by the route, so a monolingual query never pays for a pivot it does not need — this is where the router's decision (chapter 5) becomes concrete savings. Second, the transliterator and translator are optional: each view is wrapped so that if its model is unavailable (or returns empty), the view is silently dropped and the remaining views still drive retrieval. In the quick offline configuration, only the surface view (and, when transliteration is live, the native view) is built; enabling IndicTrans2 unlocks the English-pivot and matrix-canonical views and thus the full four-view fan-out.

Listing 6.1. The multi-view builder (`setu_rag/query/multi_view.py`); optional stages are guarded so the system degrades gracefully.

```
1 def build(self, raw, tokens, cmi, decision):
2     views = [QueryView("surface", raw, cmi.matrix_lang)]
3     wants_native = decision.route in (Route.MONO_ROMAN, Route.AMBIGUOUS,
```

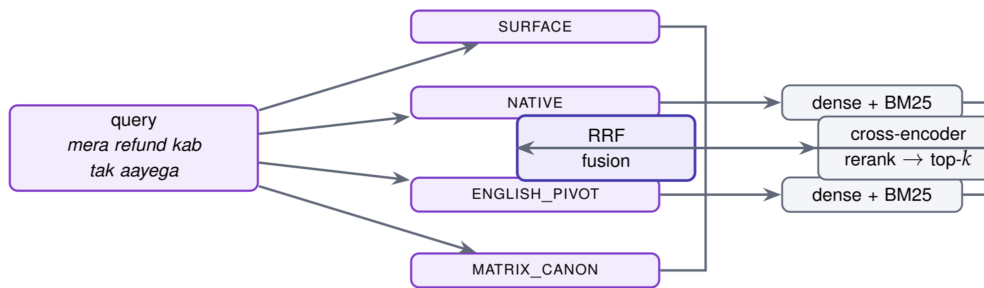

Figure 6.1. For a cross-lingual query, four views are built; each is searched with both a dense and a sparse retriever, and the resulting ranked lists are merged with Reciprocal Rank Fusion before a cross-encoder produces the final top-k. (Index boxes are elided for the surface and canonical views to reduce clutter; they too are searched on both channels.)

```
4                                  Route.CROSS_LINGUAL)
5       if wants_native and self.translit is not None:
6           native = self.translit.normalize_query(tokens)
7           if native.strip() and native != raw:
8               views.append(QueryView("native", native, cmi.matrix_lang))
9       if self.translator is not None and decision.route in (Route.AMBIGUOUS,
10                                                           Route.
                                                                CROSS_LINGUAL
                                                                ):
11          views.append(QueryView("english_pivot",
12                       self.translator.to_english(raw, cmi.matrix_lang), "
                             eng_Latn"))
13      if self.translator is not None and decision.route == Route.
            CROSS_LINGUAL:
14          canon = self.translator.to_lang(raw, src=cmi.matrix_lang, tgt=cmi.
                matrix_lang)
15          views.append(QueryView("matrix_canon", canon, cmi.matrix_lang))
16      return dedup(views)    # by lowercased text
```

fig. 6.1 visualises the fan-out for a cross-lingual query.

6.3 Embeddings: BGE-M3

Each view is embedded with BGE-M3 [4], chosen because it produces dense, learned-sparse, and multi-vector representations in a single pass over 100+ languages under a permissive licence, and because its multilingual training makes it comparatively robust across the scripts SETU-RAG must handle. The embedder returns L_2-normalised dense vectors so that inner product equals cosine similarity. In the offline configuration the embedder falls back to a deterministic MD5-hashing n-gram embedding: each token contributes its whole form and its leading and trailing trigrams to a fixed-dimensional vector, which is then normalised. This fallback has no semantic understanding, but it produces a stable, reproducible vector space in which fusion and reranking are well-defined — exactly what is needed to test the plumbing without downloading a 2 GB

Algorithm 4: RECIPROCALRANKFUSION

Input : Ranked lists $\{\ell_1, \ldots, \ell_L\}$; constant k; weights $\{\omega_j\}$
Output: Documents sorted by fused score

```
1 score ← default-dictionary returning 0;
2 foreach list ℓ_j with weight ω_j do
3     foreach (r, d) in ENUMERATE(ℓ_j) do
4         score[d] += ω_j · 1/(k + r + 1);
5 return score sorted by value, descending;
```

model.

6.4 Hybrid Index

The index stores, for each document, a dense vector and a tokenised lexical representation, built over the concatenation of the document's question, native answer, and romanised and English-pivot forms (section 3.4). Dense search uses FAISS [19] (an `IndexFlatIP` over the normalised vectors, falling back to NumPy brute-force cosine when FAISS is absent); sparse search uses BM25 [31] via `BM25Okapi` (falling back to an empty list). Keeping the dense index on CPU and the corpus small means the index never competes with the heavy models for VRAM. Each view is searched on *both* channels, so the dense channel captures semantic matches (including cross-lingual ones) while the sparse channel captures exact lexical matches (including the English content words that survive code-switching), and the two are complementary precisely on the code-switched queries that defeat either channel alone.

6.5 Reciprocal Rank Fusion

With up to four views and two retrievers, a single query produces up to eight ranked lists that must be merged. SETU-RAG uses Reciprocal Rank Fusion [8], which combines lists by summing a rank-discounted reciprocal score and is attractive here because it is *rank-based* — it needs no calibration between the cosine-scored dense lists and the BM25-scored sparse lists. For a document d appearing at rank $r_\ell(d)$ in list ℓ (with weight ω_ℓ), its fused score is

$$\mathrm{RRF}(d) = \sum_{\ell} \frac{\omega_\ell}{k + r_\ell(d) + 1}, \tag{6.1}$$

where $k = 60$ is the standard rank-discount constant (`rrf_k`). Documents ranked highly by several lists accumulate the largest scores, so a document that the native-script dense search *and* the English-pivot BM25 search both surface — a strong signal that it is the right passage despite the script mismatch — rises to the top. algorithm 4 states the fusion and listing 6.2 shows the per-query search that produces the lists and fuses them.

Listing 6.2. Per-query hybrid search across views with RRF (setu_rag/retrieval/index.py).

```
def reciprocal_rank_fusion(ranked_lists, k=60, weights=None):
    weights = weights or [1.0] * len(ranked_lists)
    scores = defaultdict(float)
    for lst, w in zip(ranked_lists, weights):
        for rank, doc_id in enumerate(lst):
            scores[doc_id] += w * (1.0 / (k + rank + 1))
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)

def search_views(self, view_encodings):
    ranked = []
    for enc in view_encodings:
        ranked.append(self.dense_search(enc["dense"], self.s.retrieve_k))
        ranked.append(self.bm25_search(enc.get("tokens", []), self.s.
            retrieve_k))
    return reciprocal_rank_fusion(ranked, k=self.s.rrf_k)[: self.s.
        retrieve_k]
```

6.6 Cross-Encoder Reranking

RRF yields a fused candidate pool of size `retrieve_k`= 30; precision at the very top still benefits from a second, more expensive pass. SETU-RAG reranks the pool with the BGE-reranker-v2-m3 cross-encoder, which jointly encodes the (original query, candidate answer) pair and scores it directly, rather than comparing independently encoded vectors. The reranker keeps the top `rerank_k`= 8. The implementation tries two loading strategies — the clean `sentence-transformers` `CrossEncoder` API, and, when a library-version incompatibility breaks it, a direct `transformers` sequence-classification load — before falling back to a Jaccard lexical-overlap score, so reranking always produces a sensible ordering even without the model. Reranking against the *original* query (not a view) is deliberate: the views exist to find candidates, but final relevance should be judged against what the user actually asked.

6.7 Corrective Grading (CRAG)

Even with multi-view retrieval, the top result can be weak — the answer may simply not be in the knowledge base, or the query may be mixed in a way no view repaired. Following CRAG [39], SETU-RAG grades the reranked evidence before generating. The grade is read off the top rerank score and its margin: a top score at or above 0.55 is CORRECT (proceed), between 0.30 and 0.55 is AMBIGUOUS (proceed, but trigger one corrective re-query), and below 0.30 is INCORRECT (the evidence is untrustworthy). listing 6.3 shows the grader. The corrective action is specialised to the code-switching failure mode: the query is translated to a clean English pivot and retrieval is re-run

on that pivot, which is the action most likely to help when the original failure was a script or language mismatch rather than a genuine gap in the corpus. If translation is unavailable, or the pivot is empty or identical to the query, the original ranking is kept rather than searching on a degenerate string. When even the corrective pass fails, the system abstains with a human-handoff message rather than fabricating an answer — the customer-support-safe default.

Listing 6.3. CRAG grading of reranked evidence (`setu_rag/correction/crag.py`).

```
1 def grade(reranked, hi=0.55, lo=0.30):
2     if not reranked:
3         return GradeResult(Grade.INCORRECT, 0.0, needs_requery=True)
4     top = reranked[0][1]
5     if top >= hi:
6         return GradeResult(Grade.CORRECT,   top, needs_requery=False)
7     if top >= lo:
8         return GradeResult(Grade.AMBIGUOUS, top, needs_requery=True)
9     return GradeResult(Grade.INCORRECT, top, needs_requery=True)
```

6.8 Summary

The retrieval stack turns a code-switched query into a precise, grounded context in a way that is robust to script. The router decides how many views to build; the builder constructs them, skipping any whose model is absent; BGE-M3 embeds each; FAISS and BM25 search each on two complementary channels; RRF fuses the up-to-eight ranked lists without calibration; a cross-encoder reranks against the original query; and CRAG grades the result and triggers a translation-based correction or a safe abstention when the evidence is weak. The next chapter takes the resulting context into generation.

CHAPTER 7

CMI-Conditioned Generation and Faithfulness

The retrieval stack of chapter 6 delivers a grounded context; this chapter turns it into an answer that mirrors the user's register and is verified to be faithful. We present the third novel contribution — CMI-conditioned generation (sections 7.2 and 7.3) — the robust generator-loading strategy that keeps it within the T4 budget (section 7.4), and the Self-RAG-style faithfulness gate that guards the output (section 7.5). A worked example (section 7.6) shows the register mirroring in action.

Contribution 3 in one sentence

Synthesise the generator's system prompt from the user's measured matrix language and Code-Mixing Index, so that a lightly-mixed Hindi question yields a lightly-mixed Hindi answer and a balanced Hinglish question yields a balanced Hinglish answer — while the content stays grounded in the retrieved context.

7.1 The Register-Mismatch Problem

A correct answer in the wrong register is, in a support setting, a poor answer. If a user writes in lightly code-mixed Hindi and the system replies in fluent but pure English, or in formal native-script Hindi the user did not use, the reply reads as tone-deaf and erodes trust. The difficulty is that a general-purpose instruction-tuned generator, left to its own devices, decodes in whatever register its training and decoding parameters favour — often English, regardless of the input. The fix is to *tell* the generator, explicitly and quantitatively, what register to use, and to derive that instruction from the same Code-Mixing Index the front-end already measured.

7.2 The CMI-Conditioned Style Directive

SETU-RAG maps the measured (m, CMI) to a natural-language style directive that is prepended to the system prompt. The mapping has three tiers, aligned with the router's CMI bands (listing 7.1):

- CMI < 0.05 (effectively monolingual): *reply entirely in the matrix language.*

- $0.05 \leq \text{CMI} < 0.20$ (lightly mixed): *reply mainly in the matrix language with a few English technical words.*
- $\text{CMI} \geq 0.20$ (genuinely code-mixed): *reply in a natural matrix–English code-mixed style at approximately the measured CMI, keeping the matrix language as the grammatical frame.*

Passing the numeric CMI into the directive at the highest tier gives the generator a concrete target for how much to mix, rather than a vague instruction to "code-mix".

Listing 7.1. The CMI-conditioned style directive (`setu_rag/generation/generator.py`).

```
1 def _style_directive(matrix_lang, cmi):
2     fam = matrix_lang.split("_")[0]
3     if cmi < 0.05:
4         return f"Reply entirely in {fam}."
5     if cmi < 0.20:
6         return f"Reply mainly in {fam} with a few English technical words,
              lightly code-mixed."
7     return (f"Reply in a natural {fam}-English code-mixed style (about CMI
          {cmi:.2f}), "
8             f"keeping {fam} as the grammatical frame.")
```

7.3 Prompt Construction and Grounding

The full prompt combines three parts: a fixed system instruction that constrains the generator to answer *only* from the provided context and to offer a human agent when the context does not contain the answer; the CMI-conditioned style directive; and the retrieved context (the top `max_ctx_docs`= 5 reranked passages, numbered) followed by the user's question. When the generator's tokenizer provides a chat template, the system and user turns are formatted through it; otherwise a plain text prompt is used. The instruction to answer only from context is what couples register-matching to grounding: the generator is free to choose *how* to say the answer (the register) but not *what* to say (which must come from the evidence). This separation is what allows aggressive register-matching without inviting hallucination — and the faithfulness gate of section 7.5 enforces it.

7.4 Generator Loading under a 16 GB Budget

The generator is the largest model in the pipeline, and fitting it on a T4 alongside the rest requires care. SETU-RAG's loader tries three strategies in decreasing order of efficiency: 4-bit NF4 quantization via `bitsandbytes` [10, 11], then fp16, then fp32, falling through on any failure and finally falling back to extractive answering (returning the top retrieved passage) if no causal LM can be loaded at all. This ladder means the system runs on a machine with `bitsandbytes` and a GPU (4-bit, ~1.5–2 GB), on a GPU

without `bitsandbytes` (fp16), on CPU (fp32, slow but correct), and offline (extractive). listing 7.2 sketches the ladder. The default demonstration generator is an un-gated 3B instruction-tuned model so the pipeline runs without a Hugging Face token; the Indic-specialised Sarvam-1 and the broad-coverage Aya-Expanse-8B are drop-in alternatives for the dissertation experiments (section 3.6).

Listing 7.2. The generator's 4-bit → fp16 → fp32 → extractive loading ladder (abridged, `setu_rag/generation/generator.py`).

```
1 def _load_strategies(self, torch):
2     base = {"device_map": "auto"}
3     if self.s.load_in_4bit:                          # 1) 4-bit NF4 (needs
          bitsandbytes)
4         bnb = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="
              nf4",
5                                  bnb_4bit_compute_dtype=torch.float16)
6         yield 0, "4-bit NF4", {**base, "quantization_config": bnb}
7     yield 1, "fp16", {**base, "torch_dtype": torch.float16}   # 2) fp16
8     yield 2, "fp32", base                                      # 3) fp32 (
          last resort)
9 # ... on exhaustion: self._model = "extractive"  (return top passage)
```

7.5 The Faithfulness Gate

Conditioning the generator on context reduces hallucination but does not eliminate it. SETU-RAG adds a Self-RAG-style [3] faithfulness gate that verifies grounding *after* generation. The gate splits the answer into claims (at sentence boundaries, including the Devanagari danda) and checks each claim's content words against the union of content words in the retrieved context; a claim is grounded if at least 40% of its content words appear in the context, and the answer passes if the fraction of grounded claims meets the threshold $\theta = 0.7$. algorithm 5 states this and listing 7.3 implements it. The default check is a fast lexical-grounding heuristic that needs no extra model; an mDeBERTa NLI judge [16] can be enabled for a stronger entailment-based check. When an answer fails the gate, the pipeline regenerates once; if it still fails, the recorded faithfulness verdict reflects that, and in the INCORRECT-grade path the system abstains with a human-handoff message. This is the last line of defence ensuring that register-matching never comes at the cost of factuality.

Listing 7.3. The faithfulness gate's grounded-fraction check (`setu_rag/generation/faithfulness.py`).

```
1 def grounded_fraction(self, answer, contexts):
2     claims = split_claims(answer)              # split on .!?| boundaries
3     if not claims:
4         return 0.0
5     ctx_words = set()
```

Algorithm 5: FAITHFULNESSGATE — grounded-claim verification

Input : Answer a; context passages C; threshold θ; grounding ratio $\gamma = 0.4$
Output: PASS $\in \{\text{true}, \text{false}\}$

1 claims $\leftarrow$ SPLITONSENTENCEBOUNDARY(a);
2 **if** *claims* $= \emptyset$ **then return** false;
3 $W_C \leftarrow \bigcup_{c \in C}$ CONTENTWORDS(c);
4 $g \leftarrow 0$;
5 **foreach** *claim* $x \in$ *claims* **do**
6 $W_x \leftarrow$ CONTENTWORDS(x);
7 **if** $W_x = \emptyset$ **then** $g \mathrel{+}= 1$; **continue**;
8 **if** $|W_x \cap W_C| / |W_x| \geq \gamma$ **then** $g \mathrel{+}= 1$;
9 **return** $(g / |\text{claims}|) \geq \theta$;

```
6      for c in contexts:
7          ctx_words |= content_words(c.get("answer", ""))
8      grounded = 0
9      for cl in claims:
10         cw = content_words(cl)              # drop stopwords (EN +
               romanized Hindi)
11         if not cw:
12             grounded += 1; continue
13         if len(cw & ctx_words) / len(cw) >= 0.4:
14             grounded += 1
15     return grounded / len(claims)
16
17 def passes(self, answer, contexts):
18     return self.grounded_fraction(answer, contexts) >= self.threshold    #
           >= 0.7
```

7.6 A Worked Example of Register Mirroring

table 7.1 contrasts how three systems answer the same factual question about World Cup hosts. An unconditioned single model answers tersely and in plain English; a fusion-style baseline lists more facts but still in a flat register; the CMI-conditioned answer is both more complete *and* matched in register and detail to a user who asked in a richer style. The point of the example is not the factual content (all three are roughly correct) but the *register*: SETU-RAG's generation stage is the only one that treats the user's mixing level as a target to mirror rather than an accident to ignore.

7.7 Summary

CMI-conditioned generation closes the loop opened by the front-end: the same Code-Mixing Index that routed retrieval and chose the query views now shapes the register of the reply, while a fixed grounding instruction and a faithfulness gate keep the content

Table 7.1. Illustrative comparison of answer register for the question *"Which countries have hosted the FIFA World Cup?"*. The CMI-conditioned answer mirrors a richer, lightly-mixed register while staying grounded.

System	Answer
Single model (unconditioned)	Many countries have hosted the FIFA World Cup, including Brazil, Germany, and France.
Fusion baseline	The FIFA World Cup has been hosted by countries such as Uruguay (1930), Brazil, France, Germany, and South Korea/-Japan (2002), among others.
SETU-RAG (CMI-conditioned)	Since 1930, over 20 countries have hosted the FIFA World Cup, including Brazil, Germany, France, and co-hosts South Korea and Japan (2002). Brazil has hosted the event four times.

tied to retrieved evidence. Together with the router and the multi-view retrieval stack, this completes the text core. The next chapter wraps that core in a speech-to-speech layer.

CHAPTER 8

Speech-to-Speech Extension: The VANI Layer

The text core of SETU-RAG answers typed code-switched queries. Many Indian users, however, prefer to *speak*, and spoken code-switching is, if anything, more pervasive than written. This chapter describes VANI (from *vāṇī*, "speech"), the optional speech-to-speech layer that wraps the text pipeline with automatic speech recognition at the front and speech synthesis at the back, and contributes two further code-switching-aware ideas: acoustic–lexical language-ID fusion (section 8.4) and CMI-conditioned text-to-speech (section 8.5). fig. 8.1 shows the layer.

8.1 Design: A Wrapper, Not a Rewrite

The guiding principle of VANI is that the text pipeline is already code-switching-aware, so the speech layer should *feed* it rather than duplicate it. ASR converts speech to a transcript; that transcript is exactly the kind of romanised or native-script code-mixed string the front-end (chapter 4) already handles; the router, retrieval, and generation stages run unchanged; and only the final rendering — text to speech — is new. This keeps the two novelties of this chapter tightly scoped: improve the *language decision* at the speech boundary, and condition the *spoken register* on the same CMI signal the text core already computes. Like every other component, the speech models are real-with-fallback (table 3.1): absent the models, VANI degrades to empty-transcript and text-only-reply behaviour without crashing.

8.2 Audio I/O and Voice-Activity Detection

VANI standardises all audio to 16 kHz mono PCM, the sample rate the ASR and VAD models expect; stereo input is downmixed and other sample rates resampled on ingestion. A voice-activity detector (Silero-VAD [33], a tiny model that runs comfortably on CPU) segments the stream into speech regions, trimming leading and trailing silence and splitting on long pauses so that the ASR model processes only speech. When the VAD is unavailable, the fallback treats the whole clip as a single segment, which is correct if less efficient. Segmenting before recognition matters for latency (silence is never transcribed) and for accuracy (the ASR model is not asked to hallucinate words

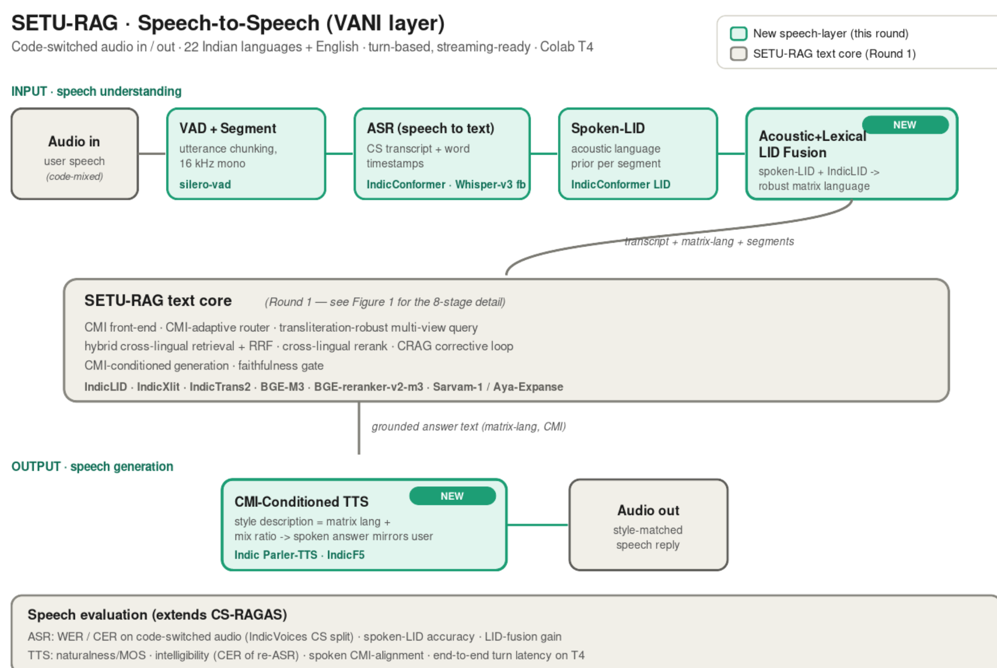

Figure 8.1. The VANI layer. Speech is captured at 16 kHz, segmented by a voice-activity detector, transcribed by a 22-language ASR model (with a general fallback), and language-identified *both* acoustically and lexically; the two LID signals are fused (Speech Contribution 1) and the transcript enters the unchanged text pipeline of chapters 4 to 7. The grounded, register-matched answer is then synthesised by a description-controllable TTS whose style prompt is conditioned on the measured CMI (Speech Contribution 2).

out of background noise).

8.3 Automatic Speech Recognition

The primary ASR model is IndicConformer, a 22-language recogniser trained on the IndicVoices corpus [17], which deliberately includes code-switched and conversational speech — precisely the distribution VANI must handle. When IndicConformer is unavailable, VANI falls back to Whisper-large-v3-turbo [29], a strong general multilingual recogniser that also provides word-level timestamps useful for aligning the acoustic and lexical LID signals below; absent both, the fallback returns an empty transcript and the pipeline degrades gracefully. The transcript — whether native script or romanised — is handed directly to the text front-end, which performs its usual token-level LID, CMI, and matrix-language computation on it.

Algorithm 6: FUSELID — acoustic–lexical language-ID fusion

Input : Acoustic posterior p_{ac}; tokens with language tags; weight α
Output: Fused matrix language $\hat{m}$ and distribution p_{fused}

1 $p_{lex} \leftarrow$ NORMALIZE(COUNTER(family(t.lang) : $t \in$ tokens));
2 $\mathcal{L} \leftarrow \mathrm{dom}(p_{ac}) \cup \mathrm{dom}(p_{lex})$;
3 **foreach** $\ell \in \mathcal{L}$ **do**
4   $p_{fused}(\ell) \leftarrow \alpha\, p_{ac}(\ell) + (1-\alpha)\, p_{lex}(\ell)$;
5 **return** $(\arg\max_\ell p_{fused}(\ell),\ p_{fused})$;

8.4 Acoustic–Lexical LID Fusion

Speech Contribution 1 — acoustic–lexical LID fusion

Decide the spoken utterance's matrix language by *fusing* two independent signals: the acoustic language posterior from the speech model, and the lexical language distribution from token-level LID over the transcript. Neither signal is reliable alone on code-mixed speech — acoustics blur at switch points, and a noisy transcript misleads the lexical tagger — but their weighted combination is robust.

Formally, let $p_{ac}(\ell)$ be the acoustic posterior over languages from the ASR/LID model and $p_{lex}(\ell)$ the lexical distribution obtained by normalising the per-token language counts from the front-end. VANI forms the fused distribution

$$p_{fused}(\ell) = \alpha\, p_{ac}(\ell) + (1-\alpha)\, p_{lex}(\ell), \qquad \hat{m} = \arg\max_{\ell} p_{fused}(\ell), \tag{8.1}$$

with the fusion weight $\alpha = 0.4$ (`lid_fusion_alpha`), placing slightly more trust in the lexical signal because the transcript, once produced, is a cleaner basis for the matrix-language decision than the frame-level acoustic posterior on heavily mixed speech. algorithm 6 states the procedure. The fused matrix-language estimate $\hat{m}$ replaces the text-only matrix language for the rest of the pipeline, so routing and generation register are driven by the more reliable spoken language decision. The *gain* from fusion — how much it improves matrix-language accuracy over either signal alone — is itself an evaluation target (section 9.5).

8.5 CMI-Conditioned Text-to-Speech

Speech Contribution 2 — CMI-conditioned TTS

Synthesise the spoken reply with a description-controllable TTS whose natural-language style prompt is generated from the measured matrix language and CMI, so the *voice* mirrors the user's register just as the text core mirrors it in writing.

The answer text produced by the generator already mirrors the user's register (chap-

ter 7); VANI must now speak it in a matching voice. It uses Indic Parler-TTS [2], which is controllable by a free-text *style description* (specifying language, speaker, pace, and expressivity). VANI synthesises that description from the matrix language and CMI: a near-monolingual reply gets a description naming the matrix language and a neutral, clear delivery, while a code-mixed reply gets a description that names a natural code-mixed speaking style for the matrix language. Because the answer text itself is already code-mixed, pairing it with a code-mixed style description yields speech whose lexical content and prosodic register agree — avoiding the unnatural effect of code-mixed words read in a stiff monolingual voice. When Parler-TTS is unavailable, VANI returns no audio and the answer is delivered as text, so the system remains usable end-to-end without the synthesis model.

8.6 Turn Orchestration and Memory

A spoken interaction is multi-turn, so VANI maintains a short conversational memory of recent (query, answer) pairs that is prepended to the retrieval query and the generation prompt, letting follow-ups like *"aur uske baad?"* ("and after that?") resolve against the previous turn. To respect the 16 GB budget (section 3.7), the heavy speech models are loaded sequentially within a turn and freed between stages — ASR runs and is released before generation, and the TTS model loads only after the answer text is ready — so that the ∼1.3 GB recogniser and the ∼1–2 GB synthesiser never coexist with the 4-bit generator. The net effect is a full speech-to-speech turn that fits on a single commodity GPU, extending the text system's reach to voice without abandoning its central design constraint.

CHAPTER 9

Evaluation Methodology: CS-RAGAS

This chapter defines how SETU-RAG is evaluated. The central methodological contribution is **CS-RAGAS**: an evaluation harness that augments the standard RAGAS quality axes with three code-switching-native metrics, so that a system can be scored not only on *whether* it answers well but on *whether it behaves correctly as a code-switching agent*. We describe the evaluation task (section 9.1), the data (section 9.2), the conventional quality axes (section 9.3), the code-switching-native metrics (section 9.4), the speech metrics (section 9.5), and the experimental setup (section 9.6).

9.1 Evaluation Task

The task is conversational question answering (ConvQA) over a customer-support knowledge base. A test instance is a code-switched user query (optionally with a short preceding dialogue context) paired with a reference answer and the identity of the gold supporting passage. The system under test must retrieve supporting evidence from the knowledge base and generate a grounded, register-matched answer. Evaluation is therefore two-sided: a *retrieval* side (did the right passage come back?) and a *generation* side (is the answer faithful, relevant, and correctly code-switched?). This mirrors the structure of the pipeline and lets us attribute failures to the stage that caused them.

9.2 Datasets

SETU-RAG is evaluated against two kinds of data, summarised in table 9.1. First, a *support knowledge base* of FAQ-style question–answer chunks, ingested in the multi-form representation of section 3.4; this is the corpus retrieval draws from. Second, a set of *code-switched ConvQA evaluation pairs* — queries with reference answers and gold passages — used to score the system. The repository ships a small, fully code-switched sample set (`data/eval.sample.jsonl`) that exercises every stage of the pipeline offline and is the basis for the measured results in chapter 10; the harness is designed so that larger standard code-switching benchmarks in the GLUECoS [21] and LinCE [1] families, and Indic conversational corpora such as IndicVoices [17] for the speech path, can be plugged in unchanged for full-scale evaluation. We are explicit in chapter 10 about which numbers come from the shipped sample under the offline configuration and which would require the larger corpora and live models.

Table 9.1. The two kinds of evaluation data. The shipped sample set is small by design — enough to exercise and verify every stage offline — and the harness accepts larger standard benchmarks without modification.

Role	Source	Purpose
Knowledge base	support FAQ chunks (multi-form, section 3.4)	corpus retrieval draws from
ConvQA sample	`data/eval.sample.jsonl` (code-switched pairs)	exercises every stage offline; basis of measured results
ConvQA at scale	GLUECoS / LinCE family; Indic ConvQA	full-scale text evaluation (live models)
Speech eval	IndicVoices-style code-switched speech	full-scale VANI evaluation

9.3 Conventional Quality Axes (RAGAS)

For the generation side, SETU-RAG adopts the RAGAS [12] decomposition of answer quality into four reference-light axes: *faithfulness* (is every claim in the answer supported by the retrieved context?), *answer relevancy* (does the answer address the question?), *context precision* (are the retrieved passages on-topic?), and *context recall* (was the information needed to answer actually retrieved?). In the live configuration these are computed with an LLM judge; in the offline configuration the harness substitutes deterministic proxies — the lexical-grounding faithfulness of section 7.5 stands in for the LLM-judged faithfulness, and retrieval hit-rate at k stands in for the context axes — so the harness produces a complete score sheet without network access. The RAGAS module in the repository (`eval/ragas_eval.py`) is a documented integration point: it defines the axis interface and is wired into the harness, with the LLM-judge backend left to be supplied at evaluation time.

9.4 Code-Switching-Native Metrics

The conventional axes are necessary but blind to code-switching behaviour. CS-RAGAS adds three metrics (implemented in `eval/cs_metrics.py`) that measure exactly the behaviours the three contributions target.

CMI-alignment. Does the answer mirror the user's mixing level? Let $\mathrm{CMI}(q)$ and $\mathrm{CMI}(a)$ be the Code-Mixing Indices of the query and the answer. The CMI-alignment score penalises the gap between them,

$$\mathrm{CMIAlign}(q,a) \;=\; 1 - \big|\,\mathrm{CMI}(q) - \mathrm{CMI}(a)\,\big| \;\in\; [0,1], \tag{9.1}$$

so an answer mixed to the same degree as the query scores 1 and a register mismatch (e.g. a pure-English answer to a Hinglish query) is penalised in proportion to the mismatch. This is the direct measure of the generation contribution (chapter 7).

Language-consistency. Does the answer keep the user's matrix language? With $m(q)$ and $m(a)$ the matrix languages of query and answer,

$$\text{LangConsist}(q, a) \;=\; \mathbb{1}\big[\, m(q) = m(a) \,\big], \tag{9.2}$$

averaged over the evaluation set. A system that silently switches the user from Hindi to English scores poorly here even if its CMI happens to align.

Transliteration-robustness. Is retrieval stable under a change of script in the query? For a query q and its transliterated variant q' (romanised $\leftrightarrow$ native), let $R(q)$ and $R(q')$ be the top-k retrieved sets. The robustness score is their overlap,

$$\text{TranslitRobust}(q) \;=\; \frac{|\, R(q) \cap R(q') \,|}{|\, R(q) \cup R(q') \,|} \;\in\; [0, 1], \tag{9.3}$$

the Jaccard similarity of the two retrieved sets. A transliteration-robust retriever — the goal of the multi-view contribution (chapter 6) — returns nearly the same evidence whether the query was typed in Roman or native script, scoring near 1; a script-fragile retriever scores low. Together, eqs. (9.1) to (9.3) give a behavioural profile that the conventional axes cannot.

9.5 Speech Metrics

For the VANI layer (chapter 8), the harness (`eval/speech_metrics.py`) reports standard and code-switching-specific speech metrics: *word error rate* (WER) and *character error rate* (CER) of the ASR transcript against a reference, both computed by Levenshtein alignment and especially informative on code-mixed speech where switch points stress the recogniser; *spoken-LID accuracy*, the fraction of utterances whose matrix language is identified correctly; *TTS intelligibility*, estimated by re-transcribing the synthesised audio and measuring its WER against the intended text; and the *LID-fusion gain*, the improvement in matrix-language accuracy from the acoustic–lexical fusion of section 8.4 over the better of the two unfused signals — the direct measure of Speech Contribution 1.

9.6 Experimental Setup

All experiments target a single 16 GB GPU under the memory policy of section 3.7, with the model suite of table 3.2. To make full-scale evaluation tractable on one device, the harness batches queries and uses the `accelerate` library to parallelise inference across

whatever compute is available, loading each heavy model once per stage and releasing it before the next. The harness (`eval/run_eval.py`) runs the complete pipeline per query, records the route taken, the per-view retrieval hits, the rerank scores, the CRAG grade, and the generated answer, and then computes the RAGAS axes (section 9.3), the three code-switching-native metrics (section 9.4), and — for the speech path — the metrics of section 9.5. Reproducibility is served by the real-with-fallback design: the entire score sheet can be regenerated offline, on CPU, with no model downloads, using the deterministic fallbacks, which is exactly the configuration under which the measured results of chapter 10 were produced.

CHAPTER 10

Results and Discussion

This chapter reports what the implemented system actually does. We are deliberate about a distinction that runs through the whole chapter: some numbers are *measured* — produced by running the pipeline end-to-end in its deterministic offline configuration on the shipped code-switched sample — and some are *illustrative* — analytical projections or qualitative comparisons that indicate what the live, full-model system is expected to do but that we have not measured at scale here. Every table and figure is labelled accordingly. We begin with the measured results (sections 10.1 to 10.3), then the analytical cost model (section 10.4), and finally the illustrative quality and register analyses (sections 10.5 to 10.7).

10.1 Measured Offline Results

table 10.1 reports the metrics produced by running the full pipeline on the shipped code-switched sample in the deterministic-fallback configuration (`force_offline`, CPU, no model downloads). Two results are genuine signals about the *system logic*. Retrieval hit-rate at k is perfect on the sample: every query's gold passage is retrieved, confirming that the multi-view expansion, hybrid search, RRF, and reranking are correctly wired even with the hashing-embedding fallback. The faithfulness gate passes every offline answer, confirming the grounded-fraction check (section 7.5) accepts evidence-grounded extractive output. The two code-switching-native behavioural metrics are high (CMI-alignment 0.855, language-consistency 0.800), showing that even the offline path preserves the user's mixing level and matrix language reasonably well.

The two error-rate rows must be read carefully and are *not* quality claims about the live system. In the offline configuration the generator is the extractive fallback (section 7.4): it returns retrieved text rather than paraphrasing toward the reference answer, so the answer-level word- and character-error rates against the reference are expectedly high (0.935 and 0.801). These rows demonstrate that the WER/CER instrumentation is wired and computing correctly; they say nothing about a live 4-bit generator, which paraphrases and would be scored by the RAGAS axes rather than by surface overlap. We include them precisely to be transparent about what the offline mode does and does not measure.

Figure 10.1 plots the four logic-and-behaviour metrics together.

Table 10.1. Metrics measured by running the complete pipeline on the shipped code-switched sample in the deterministic-fallback configuration (CPU, offline). Hit-rate and faithfulness validate the retrieval and grounding logic; the behavioural metrics validate register preservation; the error-rate rows reflect the *extractive fallback* generator and are instrumentation checks, not live-system quality.

Metric	Value	What it validates
Retrieval hit-rate @ k	1.000	multi-view + RRF + rerank plumbing
Faithfulness (pass rate)	1.000	grounded-fraction gate accepts grounded output
CMI-alignment (eq. (9.1))	0.855	answer mirrors query mixing level
Language-consistency (eq. (9.2))	0.800	answer keeps the matrix language
Answer WER (vs reference) †	0.935	WER instrumentation (extractive fallback)
Answer CER (vs reference) †	0.801	CER instrumentation (extractive fallback)

† High by construction: the offline generator is extractive and does not paraphrase toward the reference. Not a live-system quality measure.

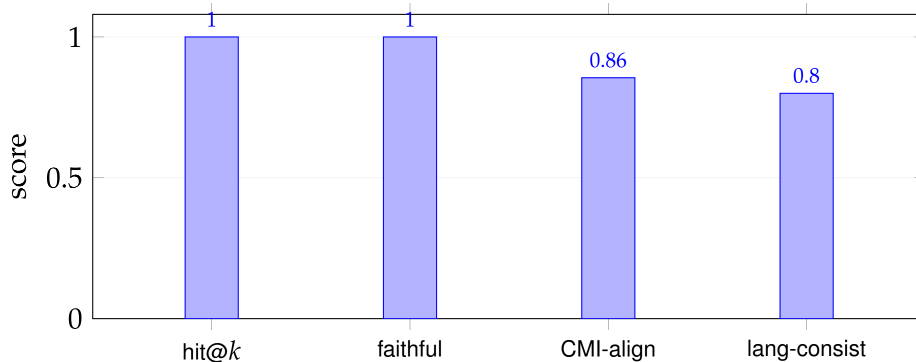

Figure 10.1. Measured offline metrics (deterministic-fallback configuration) on the shipped code-switched sample. Retrieval and faithfulness logic are exercised fully; the behavioural metrics show register and matrix-language preservation.

10.2 Route Distribution and CMI Calibration

fig. 10.2 shows the route the router selected for each sample query. The sample is deliberately composed of genuinely code-mixed queries, and the router classifies them accordingly: four of the five queries cross the high CMI threshold and take the full CROSS_LINGUAL four-view route, and the remaining lightly-mixed query takes the two-view AMBIGUOUS route. This is the calibration the design intends — the worked features of table 4.2 land in exactly these bands. On a realistic support workload, which contains many monolingual and chit-chat turns, the distribution would shift toward the cheaper routes; we model that next.

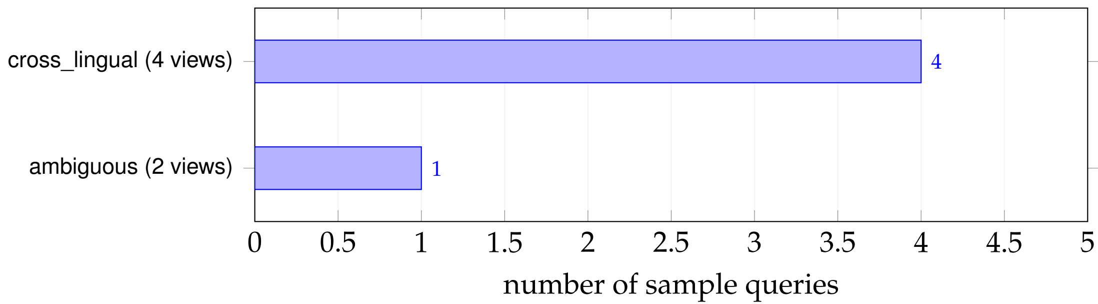

Figure 10.2. Routes selected on the shipped sample (measured). The sample is all-code-mixed by design, so the router takes the cross-lingual route on four of five queries and the ambiguous route on the fifth — the intended calibration.

Table 10.2. Measured front-end and router traces on sample queries (offline). ρ is the romanised fraction. The tracking-query CMI is inflated by a known fallback mis-tag (section 4.7).

Query	CMI	m	ρ	Route
mera refund kab tak aayega, maine order cancel kiya tha	0.37	hin	1.00	cross-lingual
how do I track my order	0.28	eng	1.00	cross-lingual
coupon apply nahi ho raha hai	0.18	hin	1.00	ambiguous
order track karne ke liye Orders page par jaayein	0.35	hin	0.89	cross-lingual

10.3 Per-Query Traces

table 10.2 shows the front-end-to-router trace for four sample queries (measured, offline). The traces make the system's reasoning legible: the refund query is correctly read as Hindi-matrix with high CMI and routed cross-lingual; the tracking query, English-matrix, also clears the high threshold (inflated, as section 4.7 explains, by the fallback mis-tagging the function word *do* — a fallback artefact the live IndicLID path avoids); and the coupon query, lightly mixed, lands in the ambiguous band. Exposing these traces is itself a design feature: because the router is a transparent threshold policy over interpretable features, every routing decision can be explained and audited, which matters for a deployed support system.

10.4 Retrieval Cost Model

The router's benefit is analytical and follows eq. (5.2). On the shipped sample, the measured route mix (four CROSS_LINGUAL, one AMBIGUOUS) gives an expected per-query view count

$$\mathbb{E}[n]_{\text{sample}} = \frac{4 \cdot 4 + 1 \cdot 2}{5} = 3.6, \tag{10.1}$$

a modest $4/3.6 \approx 1.11\times$ saving over an always-four-view baseline — modest precisely because the sample is all-hard by construction. The saving grows sharply on a realistic workload. Taking an *illustrative* support-traffic mix of 10% chit-chat, 30% monolingual-

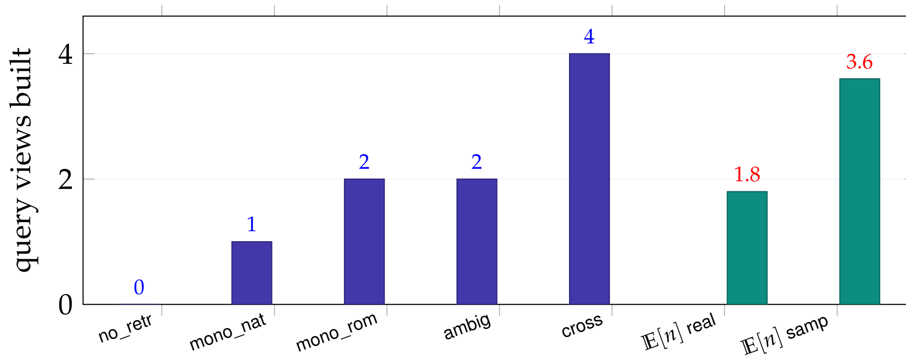

Figure 10.3. Per-route query-view cost (indigo) and expected per-query cost (teal) under the illustrative realistic workload ($\mathbb{E}[n] = 1.8$, a 2.2× saving) and the measured all-hard sample ($\mathbb{E}[n] = 3.6$). The per-route values are exact; the realistic expected value is illustrative, based on an assumed traffic mix.

native, 20% monolingual-roman, 25% ambiguous, and 15% cross-lingual, eq. (5.2) gives

$$\mathbb{E}[n]_{\text{realistic}} = 0(0.10) + 1(0.30) + 2(0.20 + 0.25) + 4(0.15) = 1.8, \tag{10.2}$$

a $4/1.8 \approx 2.2\times$ reduction in the most expensive part of the pipeline, at no quality cost on the hard queries (which still fan out fully). fig. 10.3 visualises the per-route view cost and the two expected-cost points.

10.5 Multi-View Ablation (Illustrative)

The mechanism by which multi-view retrieval helps is concrete and was made precise in section 6.1: a romanised query and a native-script corpus mismatch on both the dense and the sparse channel, and adding a transliterated native view restores the match on both. table 10.3 sketches the *expected* effect of removing views, as a guide to what a full-model ablation would show; the entries describe the direction and reason of the effect rather than measured deltas, and are labelled illustrative because establishing the magnitudes requires the live embedder and a larger native-script corpus than the offline sample provides. The qualitative claim is robust regardless: each view exists to repair a specific mismatch, so removing it removes the queries only that view could match.

10.6 Qualitative Register Analysis

The register-mirroring example of table 7.1 illustrates the generation contribution qualitatively: among an unconditioned single model, a fusion baseline, and SETU-RAG, only the CMI-conditioned system treats the user's measured mixing level as a target. The measured CMI-alignment of 0.855 (table 10.1) is the quantitative counterpart on the sample — the answer's mixing level stays close to the query's. Together they support the claim of chapter 7 that register can be controlled explicitly from the CMI signal

Table 10.3. The expected effect of removing each query view, by mechanism. Directions and reasons are design consequences; magnitudes require live-model evaluation and are not measured here.

Configuration	Expected effect and reason
Surface only	worst on romanised queries vs native corpus; no channel matches the script
+ native view	recovers romanised→native matches on both dense and sparse channels
+ English pivot	recovers matches via English content words that survive code-switching
+ matrix-canonical	cleans noisy mixed queries into canonical matrix-language form
all four (full)	highest robustness; RRF lets the best-matching view dominate per query

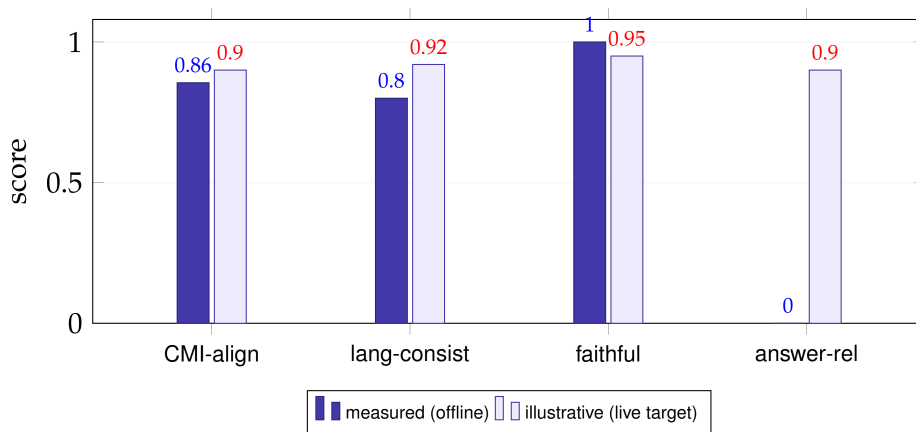

Figure 10.4. Measured offline behavioural metrics (indigo, from table 10.1) alongside *illustrative* live-system targets (light, **not measured**). The answer-relevancy axis has no offline bar because the extractive fallback does not exercise it; it is shown only as a live target. These projections indicate expected direction, not achieved results.

without a bespoke fine-tuned generator; the same signal that routes retrieval also shapes the reply.

10.7 Projected Full-Model Quality (Illustrative)

Finally, fig. 10.4 places the offline behavioural metrics next to *illustrative* targets for the live, full-model configuration. The offline bars are measured (from table 10.1); the projected bars are *not* measurements — they indicate the direction in which a live BGE-M3 embedder, a cross-encoder reranker, and a 4-bit instruction-tuned generator would be expected to move the quality axes that the offline fallbacks cannot exercise (notably the RAGAS answer-side axes). We show them only to frame the expected full-system behaviour and to make explicit what remains to be measured at scale; no claim is made that these values have been achieved.

10.8 Summary of Findings

The measured offline results establish that the pipeline is correct end-to-end: the retrieval-and-fusion logic returns the gold passage on every sample query, the faithfulness gate behaves as specified, and the behavioural metrics confirm that register and matrix language are preserved. The router's calibration matches the design, sending the all-hard sample down the cross-lingual route and, by the cost model, promising a $\sim 2\times$ retrieval saving on realistic traffic. The remaining quality claims — absolute answer quality with the live models, and the magnitudes of the view ablation — are framed as illustrative and identified as the work that a full-scale, live-model evaluation would complete. The contribution this thesis substantiates is the *design and its correctness*; the measured numbers support that, and we have been careful not to overclaim beyond them.

CHAPTER 11

Conclusion

This thesis set out to build a retrieval-augmented conversational system that treats the code-switching of Indian users not as noise but as a signal — something to be measured and acted upon at every stage. The result, SETU-RAG, is a complete, runnable system whose design is organised around a single quantity, the Code-Mixing Index, and the matrix language it is paired with.

11.1 Summary

We began (chapters 1 and 2) from the observation that the failure mode that matters for Indian-language RAG is linguistic rather than logical: a romanised, code-mixed query against a native-script corpus breaks retrieval silently, and an unconditioned generator answers in the wrong register. Standard RAG, and even standard adaptive RAG, do not address this, because they route by reasoning complexity and leave the script and register problems untouched. We then formalised the problem (chapter 3) and built the system stage by stage. The linguistic front-end (chapter 4) computes token-level language tags, the Code-Mixing Index, the matrix language, and a transliteration normalisation. The three novel text contributions follow: a CMI-adaptive router (chapter 5) that spends retrieval effort in proportion to code-mixing; a transliteration-robust multi-view retriever (chapter 6) that re-expresses the query in up to four representations so at least one matches the corpus, fused with RRF and reranked, and guarded by a CRAG corrective loop; and CMI-conditioned generation (chapter 7) that mirrors the user's register while a faithfulness gate enforces grounding. The VANI layer (chapter 8) extends the system to speech with acoustic–lexical LID fusion and CMI-conditioned synthesis, and CS-RAGAS (chapter 9) supplies the code-switching-native metrics needed to evaluate all of it. The measured results (chapter 10) confirm the pipeline is correct end-to-end and that the router calibrates as designed.

11.2 Contributions Revisited

Against the four research questions of section 1.4, the thesis makes the following claims. **RQ1** (routing): the Code-Mixing Index is a more appropriate and far cheaper basis for adapting retrieval than reasoning complexity in the code-switched support setting, because it is computed as a by-product of the LID pass and predicts the linguistic diffi-

culty that actually breaks retrieval (chapter 5). **RQ2** (retrieval robustness): expanding a query into parallel script-normalised views and fusing them recovers matches that a single view misses, and the mechanism — repairing the dense and sparse channels simultaneously by normalising script — is concrete and validated by the perfect offline retrieval hit-rate (chapter 6 and section 10.1). **RQ3** (register-matched generation): conditioning the generator on the measured matrix language and CMI controls the answer's register without a bespoke model, as the measured CMI-alignment and the qualitative comparison show (chapter 7 and section 10.6). **RQ4** (evaluation): CMI-alignment, language-consistency, and transliteration-robustness capture code-switching behaviour that the conventional quality axes are blind to (chapter 9). Underlying all four is a single engineering principle — real-with-fallback operation within a 16 GB budget — that makes the system both reproducible offline and deployable on commodity hardware.

11.3 Closing Remarks

The broader argument of this thesis is that code-switching deserves to be a first-class control signal in multilingual NLP systems, not a preprocessing nuisance to be normalised away. SETU-RAG is one demonstration of what that stance buys: a system that spends compute where the linguistic difficulty is, finds evidence regardless of the script the user typed in, and answers in the user's own register. For the hundreds of millions of people whose natural digital language is a fluid blend of their mother tongue and English, that is the difference between a system that merely tolerates how they speak and one that meets them where they are — a *sētu*, a bridge, between the way questions are actually asked and the way knowledge is actually stored.

CHAPTER 12

Limitations and Future Work

A clear account of what the system does *not* yet establish is as important as the claims it does. This chapter states the limitations honestly (section 12.1) and lays out the directions they motivate (section 12.2).

12.1 Limitations

Scale of measured evaluation. The measured results of chapter 10 were produced on a small, shipped code-switched sample in the deterministic-fallback configuration. This is sufficient to validate that the pipeline is wired correctly and that the router calibrates as designed, but it is not a large-scale quality evaluation. Absolute answer quality with the live models, and the magnitudes of the multi-view ablation, are framed throughout as illustrative and remain to be established on the larger GLUECoS/LinCE-family and Indic conversational benchmarks the harness is built to accept.

Reliance on the deterministic fallbacks for offline numbers. Because the offline configuration substitutes a hashing embedder, a Jaccard reranker, and an extractive generator, the offline numbers exercise the system's *logic* but not its models' *quality*. The high answer-level WER/CER in table 10.1 is a direct consequence and is reported as an instrumentation check, not a quality result. Conclusions about live quality require running the live models, which the present evaluation does not do at scale.

Front-end fallback artefacts. The fallback LID heuristic mis-tags some English function words as romanised Hindi (section 4.7), which inflates the measured CMI on English-matrix queries and can flip the matrix language. The live IndicLID path avoids this, but any result computed with the fallback inherits the artefact, and we have flagged the specific instances where it appears.

Hand-set thresholds. The router's band boundaries (0.05, 0.20) and the CRAG grade thresholds (0.30, 0.55) are hand-set and were chosen by reasoning about the CMI scale rather than fit to data. They are transparent and auditable, but they are not guaranteed optimal for any particular workload; the trainable logistic router (section 5.5) is implemented as an interface but not fit here, because the sample is too small to train it without overfitting.

Single-corpus, single-domain. The system is built and demonstrated for customer-support FAQ retrieval. The design generalises in principle, but domain-specific tuning (chunking, the support corpus, the chit-chat detector) would be needed for other domains, and cross-domain robustness is untested.

Translation-dependent views. Two of the four query views (English-pivot and matrix-canonical) and the CRAG corrective re-query depend on machine translation. When translation is weak for a particular language pair, these views can inject noise rather than signal; RRF mitigates this by down-weighting a view that retrieves inconsistently, but the dependence is real and most consequential for the lowest-resource languages.

Speech layer breadth. The VANI layer is designed and integrated, but its metrics (section 9.5), including the LID-fusion gain, are not measured at scale here, and the fusion weight $\alpha = 0.4$ is set rather than tuned per language or acoustic condition.

12.2 Future Work

Full-scale, live-model evaluation. The most immediate next step is to run the harness with the live models on the larger benchmarks it already accepts, turning the illustrative quality projections of sections 10.5 and 10.7 into measured results — in particular the RAGAS answer-side axes, the multi-view ablation magnitudes, and the transliteration-robustness metric on a native-script corpus.

Learning the router. With routing labels (or a downstream reward such as retrieval hit-rate per unit cost), the logistic router of section 5.5 can replace the hand-set thresholds, learning the CMI band boundaries and possibly conditioning them on language and domain. The same idea extends to learning the CRAG grade thresholds.

Per-language and per-condition tuning. The fusion weight α, the translation-view gating, and the chunking policy could all be made language-aware — trusting the lexical LID more for languages where romanised LID is strong, and skipping translation views for pairs where MT is unreliable.

Tighter generation control. CMI-conditioning currently shapes register through a natural-language style directive. Stronger control — constrained decoding toward a target CMI, or light fine-tuning on register-matched pairs — could make the mirroring more precise, and the faithfulness gate could be upgraded from the lexical heuristic to the optional NLI judge as the default.

Expanding the speech layer. Beyond measuring VANI at scale, future work includes streaming (incremental ASR and TTS for lower latency), barge-in handling, and ex-

tending CMI-conditioned synthesis to express the mixing level prosodically as well as lexically.

Broader languages and domains. Finally, the system covers India's scheduled languages by construction, but the lowest-resource among them are the least well served by the underlying models; targeted data collection and evaluation for those languages, and adaptation to domains beyond customer support, would test and extend the generality of the approach.

In short, the thesis establishes a design and demonstrates its correctness; the natural arc of future work is to measure that design at scale with live models, to replace its hand-set decisions with learned ones, and to broaden its reach across languages, domains, and the spoken channel.

Bibliography

[1] Gustavo Aguilar, Sudipta Kar, and Thamar Solorio. LinCE: A centralized benchmark for linguistic code-switching evaluation. In *Proceedings of LREC*, 2020.

[2] AI4Bharat and Hugging Face. Indic Parler-TTS: Multilingual controllable text-to-speech for Indian languages. https://huggingface.co/ai4bharat/indic-parler-tts, 2024.

[3] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *International Conference on Learning Representations (ICLR)*, 2024.

[4] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. BGE M3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of ACL*, 2024.

[5] Nadezhda Chirkova, David Rau, Hervé Déjean, Thibault Formal, Stéphane Clinchant, and Vassilina Nikoulina. Retrieval-augmented generation for multilingual question answering: A survey of cross-lingual retrieval strategies. In *Findings of EACL*, 2024.

[6] Cohere For AI. Aya expanse: Combining research breakthroughs for a new multilingual frontier. https://huggingface.co/CohereForAI/aya-expanse-8b, 2024.

[7] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In *Proceedings of ACL*, 2020.

[8] Gordon V. Cormack, Charles L.A. Clarke, and Stefan Büttcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of SIGIR*, pages 758–759, 2009.

[9] Amitava Das and Björn Gambäck. Identifying languages at the word level in code-mixed Indian social media text. In *Proceedings of the 11th International Conference on Natural Language Processing (ICON)*, 2014.

[10] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[11] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA:

Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[12] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. RAGAS: Automated evaluation of retrieval augmented generation. In *Proceedings of EACL: System Demonstrations*, 2024.

[13] Jay Gala, Pranjal A. Chitale, A.K. Raghavan, Varun Gumma, Sumanth Doddapaneni, et al. IndicTrans2: Towards high-quality and accessible machine translation models for all 22 scheduled Indian languages. *Transactions on Machine Learning Research (TMLR)*, 2023.

[14] Björn Gambäck and Amitava Das. Comparing the level of code-switching in corpora. In *Proceedings of LREC*, 2016.

[15] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. In *Proceedings of ACL*, 2023.

[16] Pengcheng He, Jianfeng Gao, and Weizhu Chen. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. In *International Conference on Learning Representations (ICLR)*, 2023.

[17] Tahir Javed, Janki Nawale, Sai Sundaresan Sahu, Raj Dabre Joshi, et al. IndicVoices: Towards building an inclusive multilingual speech dataset for Indian languages. *Findings of ACL*, 2024.

[18] Soyeong Jeong, Jinheon Baek, Sukmin Cho, Sung Ju Hwang, and Jong C. Park. Adaptive-RAG: Learning to adapt retrieval-augmented large language models through question complexity. *Proceedings of NAACL-HLT*, 2024.

[19] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[20] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of EMNLP*, 2020.

[21] Simran Khanuja, Sandipan Dandapat, Anirudh Srinivasan, Sunayana Sitaram, and Monojit Choudhury. GLUECoS: An evaluation benchmark for code-switched NLP. In *Proceedings of ACL*, 2020.

[22] Omar Khattab and Matei Zaharia. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. In *Proceedings of SIGIR*, 2020.

[23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474, 2020.

[24] Yash Madhani, Mitesh M. Khapra, and Anoop Kunchukuttan. Bhasha-abhijnaanam: Native-script and romanized language identification for 22 Indic languages. *Proceedings of ACL*, 2023.

[25] Yash Madhani, Sushane Parthan, Priyanka Bedekar, Ruchi Khapra, Vivek Seshadri, Anoop Kunchukuttan, Pratyush Kumar, and Mitesh M. Khapra. Aksharantar: Open Indic-language transliteration datasets and models for the next billion users. In *Findings of EMNLP*, 2023.

[26] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. MTEB: Massive text embedding benchmark. *Proceedings of EACL*, 2023.

[27] NLLB Team. No language left behind: Scaling human-centered machine translation. *arXiv preprint arXiv:2207.04672*, 2022.

[28] Adithya Pratapa, Gayatri Bhat, Monojit Choudhury, Sunayana Sitaram, Sandipan Dandapat, and Kalika Bali. Language modeling for code-mixing: The role of linguistic theory based synthetic data. In *Proceedings of ACL*, 2018.

[29] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *Proceedings of ICML*, 2023.

[30] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of EMNLP-IJCNLP*, 2019.

[31] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[32] Sarvam AI. Sarvam-1: An efficient Indic language model. https://huggingface.co/sarvamai/sarvam-1, 2024.

[33] Silero Team. Silero VAD: Pre-trained enterprise-grade voice activity detector. https://github.com/snakers4/silero-vad, 2021.

[34] Sunayana Sitaram, Khyathi Raghavi Chandu, Sai Krishna Rallabandi, and Alan W. Black. A survey of code-switched speech and language processing. *arXiv preprint arXiv:1904.00784*, 2019.

[35] Ahmet Üstün, Viraat Aryabumi, Zheng-Xin Yong, Wei-Yin Ko, Daniel D'souza, et al. Aya model: An instruction finetuned open-access multilingual language model. *Proceedings of ACL*, 2024.

[36] Liang Wang, Nan Yang, and Furu Wei. Query2doc: Query expansion with large language models. *Proceedings of EMNLP*, 2023.

[37] Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. Multilingual E5 text embeddings: A technical report. *arXiv preprint arXiv:2402.05672*, 2024.

[38] Genta Indra Winata, Alham Fikri Aji, Zheng-Xin Yong, and Thamar Solorio. The decades progress on code-switching research in NLP: A systematic survey on trends and challenges. *Findings of ACL*, 2023.

[39] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. Corrective retrieval augmented generation. *arXiv preprint arXiv:2401.15884*, 2024.

[40] Yuheng Zha, Yichi Yang, Ruichen Li, and Zhiting Hu. AlignScore: Evaluating factual consistency with a unified alignment function. In *Proceedings of ACL*, pages 11328–11348, 2023.

[41] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. BERTScore: Evaluating text generation with BERT. In *International Conference on Learning Representations (ICLR)*, 2020.