AI Admission Agent

By

Rakymzhan Zhabagin, Adilet Zhumabay

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering
Univerity of Bologna

6699 — Intelligent Embedded Systems

July 2026
Cesena

Abstract

This project implements a domain-bounded conversational assistant for prospective students who need practical guidance about applying to Italian universities and handling adjacent study procedures such as student visas, residence permits, scholarships, and admission documents. The system is exposed primarily through Telegram, but its internal design is split into a Go bot layer and a Python retrieval service. The Go layer handles Telegram updates, access control, optional Redis-backed conversation memory, and optional event publication. The Python service performs guardrail checks, intent classification, retrieval-query clarification, document search, grounded answer generation, and attachment-handling logic.[1],[2],[3],[4]

The repository shows a deliberate attempt to keep the assistant narrow in scope. Queries related to study abroad in Italy are routed toward retrieval over a curated document corpus, while unrelated or unsafe requests are refused before answer generation. The corpus itself is practical rather than encyclopedic: it contains visa, residence-permit, DSU scholarship, CV, motivation-letter, and recommendation-letter material in English, Russian, and partly Kazakh, which indicates a multilingual support goal for applicants who may not be comfortable using only English or Italian.[5],[6],[7]

The main contribution of the project is not a novel model, but an end-to-end engineering design that combines access-controlled messaging, document-grounded assistance, lightweight conversation memory, and operational tooling for dataset refresh and administration. At the same time, the repository also reveals important limits: the corpus is narrow, official procedures are time-sensitive, and the system depends on careful prompt design and retrieval quality to remain reliable. For that reason, the assistant is best understood as an informational support tool rather than a legal or institutional authority.[3],[8],[1],[2]

List of Figures

FIGURE Page

1.1 Telegram bot answering a typical in-scope question about the Italian student visa process and returning grounded guidance with a supporting document. . 2

2.1 Example of refusal or redirection when the user asks for an out-of-scope request unrelated to education abroad. 5

4.1 Overall architecture of the AI admission agent. 10
4.2 AI Admission Agent Detailed Class Diagram. 12
4.3 Administrative dashboard used for access control and operational monitoring of the system. 13
4.4 PostgreSQL database schema used for access control, conversation storage, message classification, and usage-event persistence. 14
4.5 Processing stages used for a retrieval-grounded answer. 15

5.1 Example of context-aware follow-up handling in Telegram, where the bot interprets a short message using recent conversation state. 21
5.2 Public landing page of the system. 22
5.3 Equivalent ISEE calculator page provided as a public web interface for student financial estimation. 23

6.1 Refusal of an unsafe request involving falsification or misuse of visa-related or admission-related documents. 25

8.1 Containerized runtime and supporting infrastructure. 33

Table of Contents

Abstract **i**

List of Figures **ii**

1 Introduction **1**

2 Requirements and Scope **3**

2.1 Functional Requirements 3

2.2 Non-Functional Requirements 4

2.3 In-Scope and Out-of-Scope Behavior 4

2.4 Accepted and Rejected Request Types 4

2.5 Why Coding Requests Are Refused 5

3 Domain and Regulatory Context **7**

3.1 High-Level Student Path 7

3.2 Pre-Enrolment Through Universitaly 7

3.3 Student Visa Process 8

3.4 Residence Permit After Arrival 8

3.5 Role Boundaries 9

3.6 Stable Facts Versus Time-Sensitive Facts 9

4 Architecture and Design **10**

4.1 Overall Architecture 10

4.2 Telegram Interface 11

4.3 Backend Services and Their Responsibilities 11

4.3.1 Class Diagram and Structural Design 11

4.4 RAG Flow 13

4.5 Classification and Routing Design 16
4.6 Redis Memory and Session Design 16
4.7 Refusal and Guardrail Path 16
4.8 Design Rationale and Trade-Offs 17

5 Implementation **18**
5.1 Languages, Frameworks, and APIs 18
5.2 Telegram Integration Approach 18
5.3 Document and Data Ingestion 19
5.4 Retrieval and Prompting 19
5.5 Redis Data Structures and TTL Assumptions 20
5.6 Conversation State Handling 20
5.7 Error Handling and Logging 20
5.8 Implementation Observations 21

6 Safety, RAG, Memory, and Guardrails **24**
6.1 Domain-Bounded Behavior 24
6.2 Guardrail Design 24
6.3 Unsafe-Request Handling 25
6.4 Source-Grounded Answering 26
6.5 Memory Usage and Its Purpose 26
6.6 Risks of Stale Information 26
6.7 Prompt Injection and Misuse Considerations 27
6.8 Why Source Citation Matters 27

7 Validation, Limitations, and Evaluation **28**
7.1 Available Tests 28
7.2 Manual Testing Scenarios Implied by the Suite 29
7.3 Key Limitations 29
7.4 Hallucination, Retrieval, and Memory Risks 30
7.5 Realistic Evaluation Plan and Future Work 30

8 Deployment and User Guide **32**
8.1 Local and Containerized Runtime Layout 32
8.2 Environment Variable Categories 33
8.2.1 Go Bot 33
8.2.2 Python RAG API 34

8.2.3 Admin Panel 34
8.3 High-Level Telegram Bot Setup 35
8.4 Data and Artifact Setup 35
8.5 Redis Setup 35
8.6 Testing Example Questions 36
8.7 Concise End-User Guide 36

9 Self-Evaluation and Future Work **37**
9.1 What Is Already Strong 37
9.2 What Remains Fragile 37
9.3 Architectural Quality 38
9.4 Future Work 38
9.5 Final Assessment 39

A Notes **40**

Bibliography **45**

Chapter 1

Introduction

The central idea of this project is to provide a focused conversational assistant for admission-related questions about studying in Italy. Instead of trying to behave like a general-purpose chatbot, the system narrows itself to a bounded set of educational and administrative topics: university application steps, student visas, residence permits, DSU scholarship guidance, and preparation of common admission documents such as CVs, motivation letters, and recommendation letters.[3],[11],[2]

This bounded scope is visible directly in the corpus and in the routing logic. The project stores domain material as structured markdown derived from source documents, then retrieves from that corpus when the user asks factual or procedural questions. The available document set is practical and student-facing rather than academic in the strict sense: it explains how to prepare a CV, what a motivation letter should contain, what kind of recommendation letters are useful, how DSU scholarship support works, and how visa or residence-permit procedures are described in the maintained guides.[5],[6],[12],[13],[14],[15]

The likely target users are international applicants who need help navigating Italian higher-education procedures in a multilingual setting. The repository does not contain a formal persona document, but the language coverage is revealing. Several corpus items exist in English, Russian, and Kazakh, and the query-routing prompts explicitly distinguish among en, ru, and kk. This strongly suggests that the assistant was designed for users who may move between these languages, which is consistent with a Central Asian student-support use case, especially for Kazakhstan-based or Russian-speaking applicants.[11],[16],[17]

Italy is a suitable bounded domain for such a system because the student journey is structured but information-heavy. A prospective student must identify a programme,

verify entry requirements, interact with a university, use Universitaly for pre-enrolment where applicable, seek a study visa through the competent diplomatic channel, and then comply with residence obligations after arrival. These steps are not identical for every university or region, but they are regular enough to justify a domain assistant as long as the assistant remains careful about scope and source freshness.[1],[3],[2]

The project's value proposition is therefore operational rather than spectacular. It reduces friction in repetitive informational interactions, provides multilingual answers grounded in a small document base, remembers recent conversational context, and can send supporting files when the retrieval pipeline identifies a relevant source. The benefit is not that the assistant replaces universities, embassies, or Questure, but that it can help users orient themselves before approaching those authorities or official websites.[19],[20],[21]

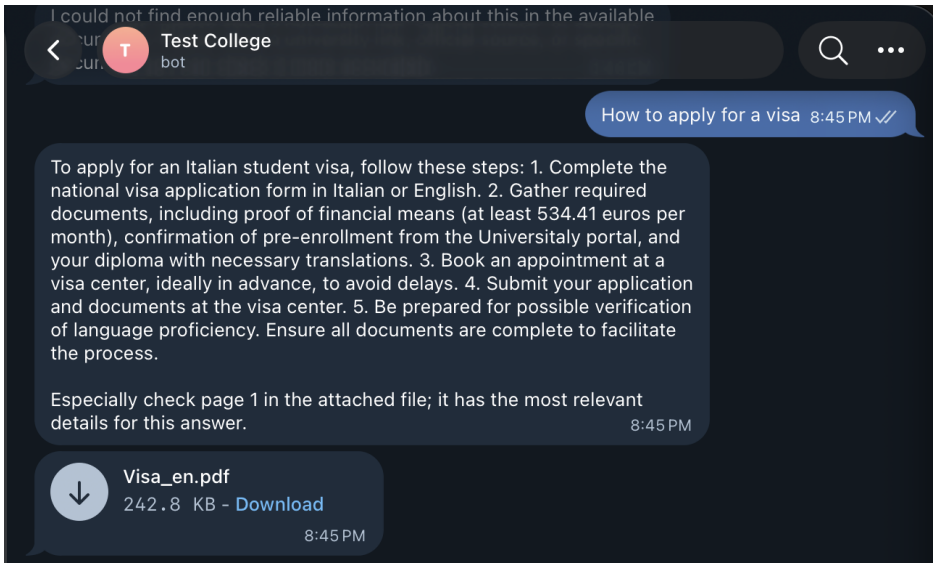

Figure 1.1: Telegram bot answering a typical in-scope question about the Italian student visa process and returning grounded guidance with a supporting document.

The system boundaries are also clear. It is not a universal search engine, not a legal advisor, not a travel planner, and not a coding assistant. The guardrail and classifier prompts explicitly reject out-of-domain tasks such as programming help, tourism planning, or unsafe requests involving document fraud or evasion of official procedures. That decision is conceptually important: the project chooses reliability through scope restriction rather than apparent versatility.[11],[16]

Chapter 2

Requirements and Scope

The repository does not contain a standalone requirements specification, so the requirements in this chapter are reconstructed from the actual code paths, configuration, prompt rules, and tests. This means the chapter describes implemented behavior and strongly supported intent, not an aspirational product roadmap.[22],[11],[23]

2.1 Functional Requirements

The implemented system supports the following core functions.

- Accept text questions from Telegram users and return an answer, optionally with a supporting file.
- Enforce access control before any answer is generated, using a users table in PostgreSQL and optional Redis caching of access decisions.
- Forward eligible questions from the Go bot to a local Python API together with a conversation identifier and an optional preferred name.
- Run a guardrail stage before classification so that unrelated or unsafe requests can be rejected early.
- Classify the request into greeting, chitchat, factual question, procedure, comparison, or out-of-domain behavior, and route accordingly.
- Retrieve relevant document chunks for in-scope questions, synthesize a grounded response, and choose a supporting file when appropriate.
- Maintain short conversation history in Redis so that follow-up queries such as ”yes”, ”how”, or ”send it” can be interpreted in context.
- Publish turn events to RabbitMQ and persist them to PostgreSQL through a

worker process when that optional pipeline is enabled.
- Provide an admin panel for user access management, usage inspection, and LightRAG artifact operations.

2.2 Non-Functional Requirements

Several non-functional requirements are also evident from the repository.

- The bot should degrade gracefully when optional subsystems are unavailable. For example, the Go runtime logs warnings and continues without Redis cache, conversation memory, or RabbitMQ turn events if those integrations fail at startup.[1]
- The assistant should stay narrow in scope and prefer refusal over unsupported improvisation.[11]
- The RAG service should preserve conversational continuity across short follow-ups without storing unbounded history; both the Go and Python memory layers cap retained items.[30],[31]
- The system should be operable with container-based deployment and branch-based CI/CD.[35],[36],[37]
- The answer path should preserve multilingual behavior across English, Russian, and Kazakh where possible.[11],[16]

2.3 In-Scope and Out-of-Scope Behavior

The intended scope is broader than "university admission" in the narrow sense, but still clearly bounded. In scope are admission procedures, required documents, CVs, recommendation letters, motivation letters, DSU scholarship questions, student visa questions, student residence-permit questions, and related study-abroad comparisons.[11],[11],[38]

Out of scope are generic software-development help, travel planning, work-visa advice, unrelated general knowledge, and unsafe activities such as document falsification or bypassing immigration rules. The prompts and tests are explicit on this point. For example, programming queries are expected to become `OUT_OF_DOMAIN`, and unsafe requests are expected to be blocked before classification or retrieval.[11],[39],[23]

2.4 Accepted and Rejected Request Types

Examples of accepted requests supported by the code and tests include:

Accepted requests

‘‘How to apply for visa?’’
‘‘What photo do I need for visa?’’
‘‘How to apply for residence permit?’’
‘‘What is DSU?’’
‘‘How should I prepare a CV for university admission in Italy?’’
‘‘Send recommendation letter example for Italian university.’’

Examples of rejected or redirected requests include:

Rejected or redirected requests

‘‘How do I build a Docker image?’’
‘‘Find cheap flights to Milan.’’
requests to fake or forge documents
requests to evade immigration checks or lie in applications

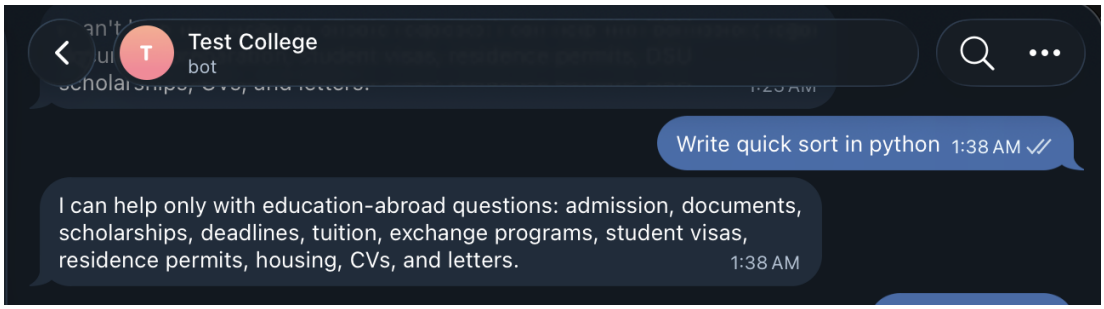

Figure 2.1: Example of refusal or redirection when the user asks for an out-of-scope request unrelated to education abroad.

2.5 Why Coding Requests Are Refused

The project is intentionally domain-bounded. Coding requests are refused not because the underlying model would be incapable of attempting them, but because they would violate the assistant’s safety and product boundary. In the classifier prompt, programming is explicitly treated as unrelated to education abroad, and the tests verify that such queries should be redirected out of scope rather than answered as normal assistance.[11],[23]

This is a sound systems decision. A student-admission assistant becomes less trustworthy if it pretends to be an expert in arbitrary topics. The repository therefore favors specialization, even at the cost of rejecting some seemingly easy requests.

Chapter 3

Domain and Regulatory Context

The assistant operates in a domain where procedural accuracy matters. Admission to Italian higher education is not a single transaction but a chain of related steps involving universities, the national Universitaly platform, diplomatic or consular authorities, and post-arrival immigration procedures. For that reason, a useful student-support bot must distinguish between stable structural facts and information that varies by institution, region, or academic year.[1],[4],[5]

3.1 High-Level Student Path

Universitaly describes a "10-steps" path to study in Italy. The first stages are to identify a programme, verify entry requirements, and check language expectations. The official page emphasizes that applicants should choose a programme, verify whether they are applying to a first-cycle, single-cycle, or second-cycle course, and then confirm the specific requirements directly with the chosen institution.[1]

This structure fits the repository corpus well. The local guides and prompts repeatedly assume that a user may need help with programme choice, admission documents, visa preparation, and later administrative obligations. The corpus is especially practical for users who are already committed to the Italy pathway and need procedural guidance rather than broad comparative counselling.[13],[14],[15]

3.2 Pre-Enrolment Through Universitaly

The repository's visa guide states that applicants for Italian university study should submit a preliminary application through the Universitaly portal before the visa stage,

and the official Universitaly materials confirm the centrality of pre-enrolment procedures for international students.5,40 At the same time, the official "Procedures for entry, residency and enrolment of international students" page makes an important institutional distinction: higher-education institutions evaluate admission and qualifications, while the diplomatic or consular representation remains solely responsible for the final visa decision.[4]

This matters for system design. A student-support bot can explain the relationship between pre-enrolment and later visa steps, but it should not imply that a validated pre-enrolment automatically guarantees a visa appointment or a positive visa outcome.

3.3 Student Visa Process

The local corpus describes the student visa as a study-related long-stay process for university enrolment, including pre-enrolment, document preparation, and submission through the competent channel.5 The official visa portal reinforces the high-level legal frame: long-stay visas are for stays above 90 days, and non-EU citizens intending to stay for such periods generally need a national long-stay visa. The same portal also distinguishes the `STUDIO` purpose from other visa categories, which is relevant because this assistant should remain on the study track and not drift into tourism or work-visa advice.[5]

Because visa practice is time-sensitive, the bot should treat the following as variables that need official checking:

- document lists
- financial-threshold details
- appointment logistics
- country-specific consular outsourcing channels
- yearly deadlines[4],[5]

3.4 Residence Permit After Arrival

The repository's residence-permit guide states that the applicant should apply for the permit within eight days after arrival and describes a sequence involving Poste Italiane, fingerprints, and the Questura.6 The official visa portal aligns with the timing principle: after entering Italy for a long stay, the foreign citizen must comply with residence obligations and apply for a permit of stay within the legally indicated period.[5]

This is a good example of why the assistant should cite or point users toward exact sources. The broad structure is stable, but details such as which office to use, how long appointments take, and which supporting documents are needed may vary in practice.

3.5 Role Boundaries

The system should keep the following role boundaries explicit.

- The university evaluates programme-specific admission conditions and recognition for access to its own courses.[1],[4]
- Universitaly serves as a national information and procedure platform, including international-student guidance and pre-enrolment support.
- The embassy or consulate decides visa issuance; official Universitaly guidance explicitly states that final visa release is the exclusive competence of the diplomatic-consular representation.
- The Questura and related immigration institutions handle post-arrival stay formalities.
- The student remains responsible for document accuracy, timing, and compliance with the relevant authorities.

3.6 Stable Facts Versus Time-Sensitive Facts

Stable domain facts include the existence of programme selection, entry requirements, language requirements, study-visa procedures, and residence-permit obligations after arrival.[1]

Time-sensitive or institution-specific facts include:

- programme deadlines
- admissions tests and interview schedules
- scholarship amounts and regional DSU differences
- exact visa documentation and proof-of-funds expectations
- local Questura waiting times
- university-specific document templates and language policies [12]

The assistant is therefore best positioned as an informational guide that explains the process and helps users navigate documents, while still directing them to official sources when the answer depends on current rules or institution-specific practice.

Chapter 4

Architecture and Design

The repository implements a distributed but still compact architecture. Its main runtime path starts in Telegram, passes through a Go bot service, reaches a Python RAG API for reasoning and retrieval, and optionally persists interaction traces through RabbitMQ and PostgreSQL. A separate FastAPI admin panel shares the database domain and provides operational controls.[1]

4.1 Overall Architecture

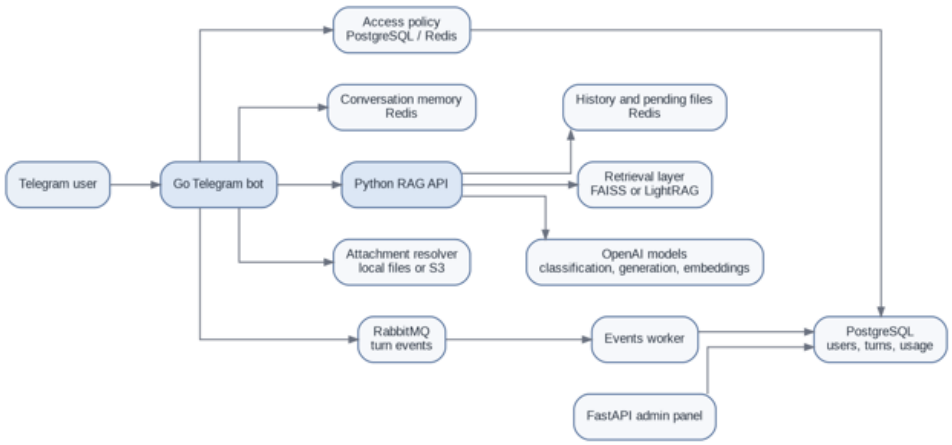

Figure 4.1: Overall architecture of the AI admission agent.

The Go service is intentionally thin. It owns Telegram integration, access control, local conversation-key management, attachment resolution, and optional event publication, but it delegates question understanding and grounded answering to the Python API.[24],[19]

4.2 Telegram Interface

The Telegram layer uses the official Bot API model through the Go `go-telegram/bot` package. The handlers register `/start`, `/help`, and `/randompic`, while normal text messages go through a default handler. Unsupported input such as photos or non-text payloads is rejected with a language-aware message rather than being silently ignored.[24]

This adapter design is sensible because it keeps Telegram-specific concerns separate from the domain logic. It also allows the Go layer to expose transient progress messages while the downstream query is still being processed.[44]

4.3 Backend Services and Their Responsibilities

The key backend responsibilities are divided as follows.

- Go bot: access enforcement, Telegram update handling, request forwarding, local attachment delivery, optional Redis cache, optional Redis memory, optional turn-event publication.[1],[24],[30]
- Python RAG API: guardrails, classification, query clarification, retrieval, grounding prompts, fallback handling, file-offer logic, and usage-event estimation.[3],[11],[28]
- Worker: asynchronous persistence of turn events into relational tables for later inspection.[42],[33]
- Admin panel: operational visibility and control over users, conversations, usage events, and LightRAG build jobs.[43],[8]

4.3.1 Class Diagram and Structural Design

To complement the component-level architecture, the object-oriented design and domain entity relationships are mapped in the class diagram below (Figure 4.2).

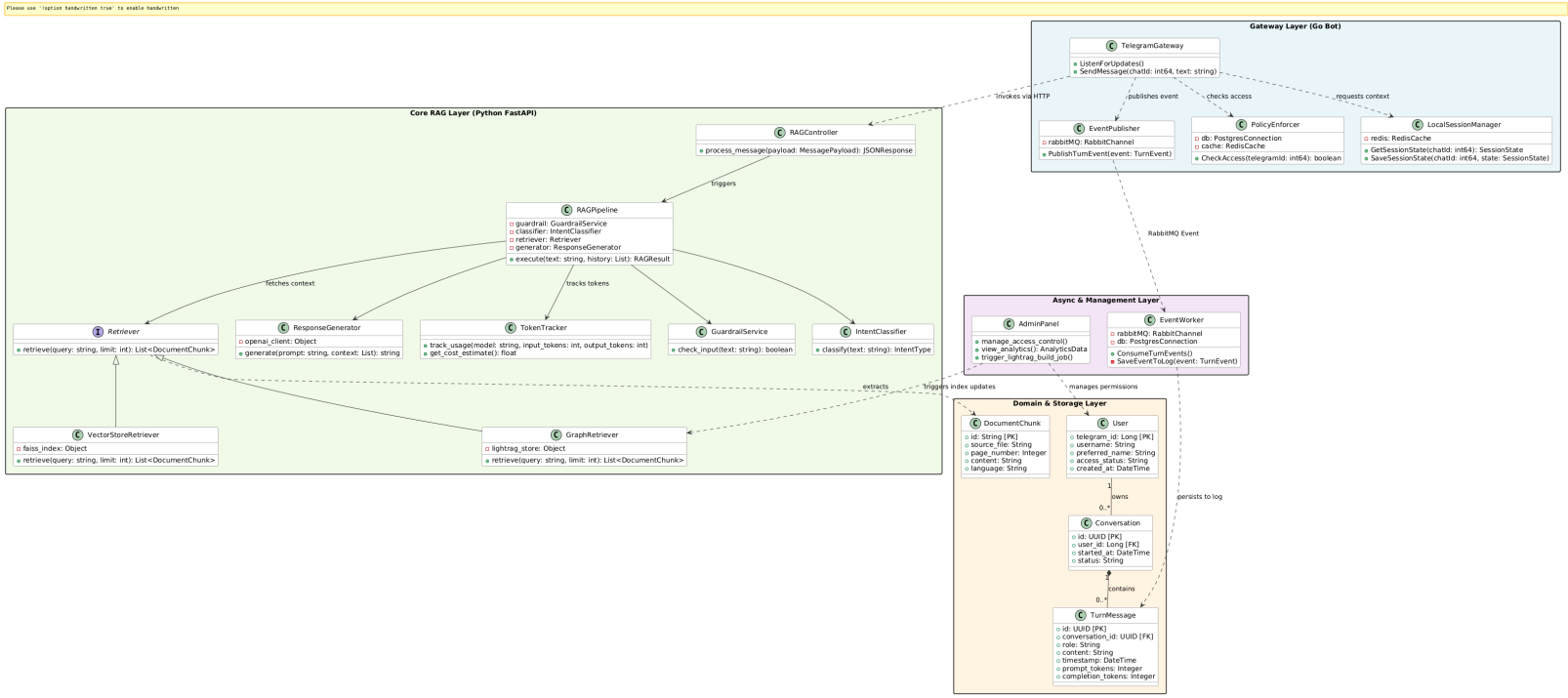

Figure 4.2: AI Admission Agent Detailed Class Diagram.

The structural design is partitioned into four clear functional layers:

Gateway Layer (Go Bot): Handles the Telegram interface, checks user permissions, caches session IDs via Redis, and pushes analytics asynchronously.

Core RAG Layer (FastAPI): Coordinates request evaluation through `RAGPipeline`, isolates OpenAI interactions, and encapsulates FAISS or LightRAG search behind a shared retriever interface.

Domain & Storage Layer: Maps the data schema where a single `User` maintains multiple contextual `Conversation` records, each composing a structured list of historical `TurnMessage` logs.

Async & Management Layer: Contains the background queue consumer for PostgreSQL telemetry logs, alongside the operational `AdminPanel`.

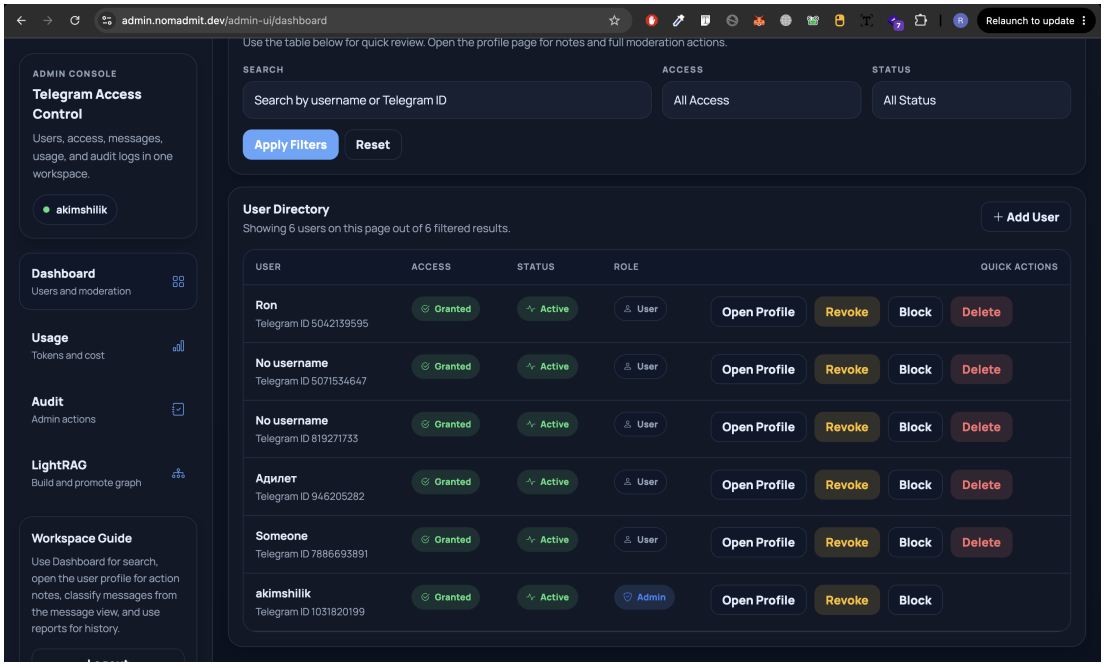

Figure 4.3: Administrative dashboard used for access control and operational monitoring of the system.

4.4 RAG Flow

The RAG path in the Python service is more layered than a simple ”embed and answer” design.

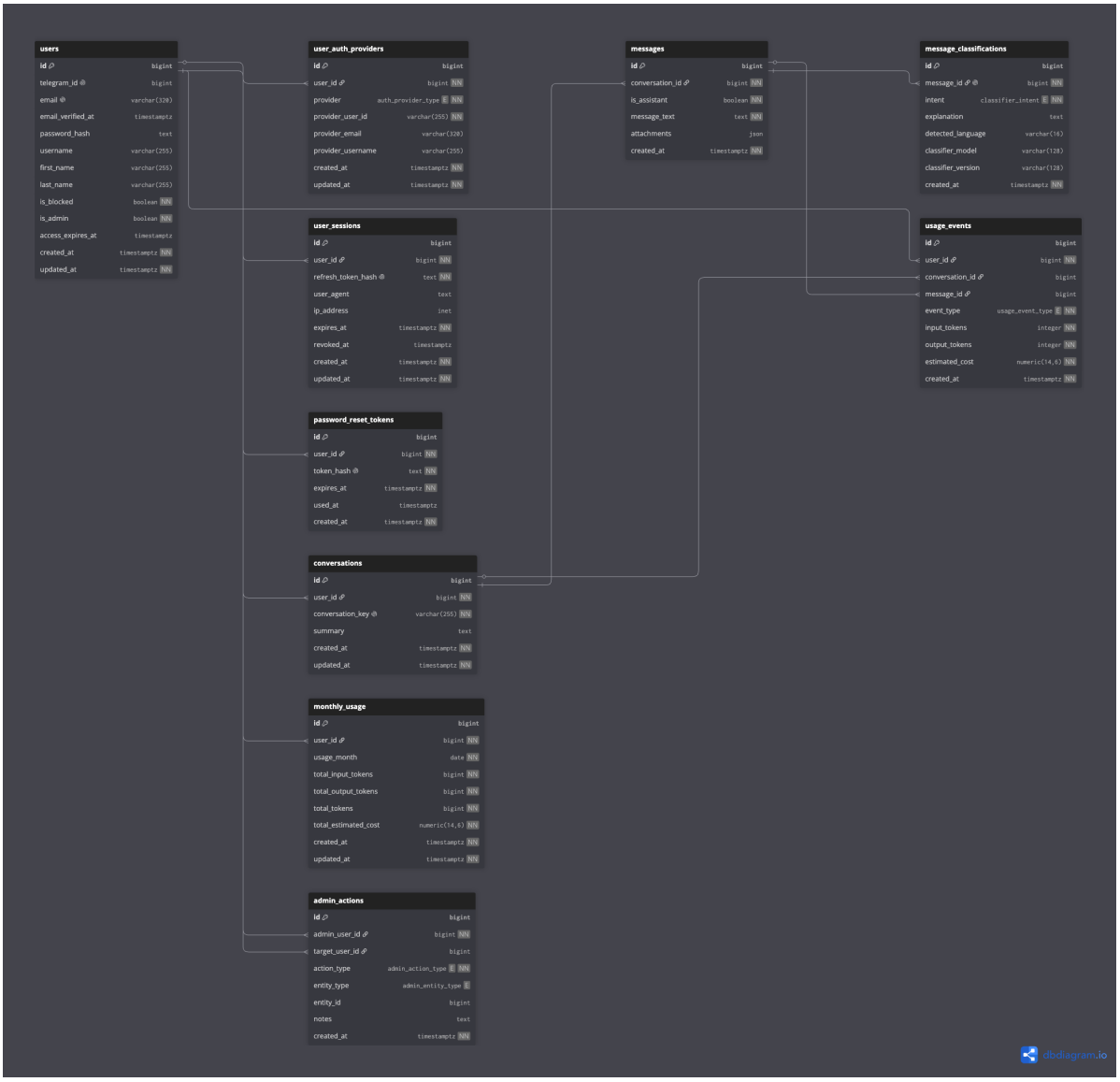

Figure 4.4: PostgreSQL database schema used for access control, conversation storage, message classification, and usage-event persistence.

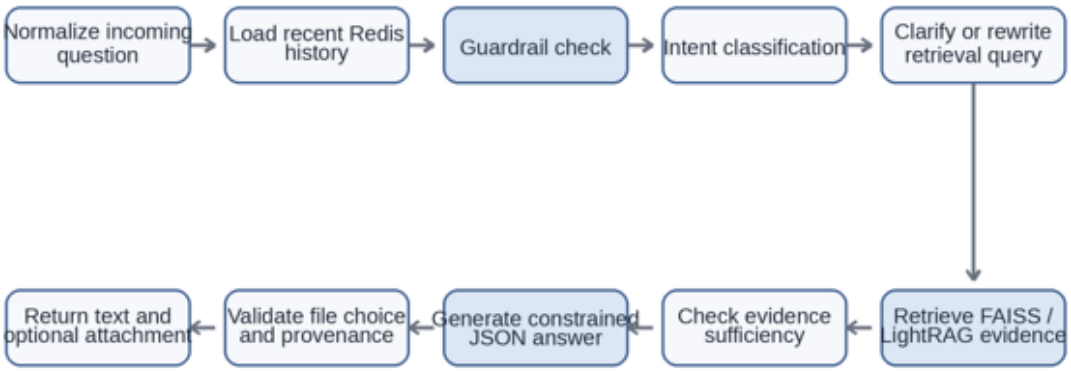

Figure 4.5: Processing stages used for a retrieval-grounded answer.

1. The service normalizes the latest query and optionally loads recent history from Redis.[3],[46]
2. A first-pass guardrail checks whether the input is allowed, unsafe, out of scope, or too elliptical to judge without context.[11]
3. If needed, the service retries the guardrail with history, then runs intent classification.[3],[11]
4. For retrieval-oriented intents, a clarity stage decides whether the query is already clear, needs rewriting, or needs a clarifying question.
5. The chosen retrieval backend performs search over FAISS or LightRAG artifacts, optionally refining results toward better file-level focus.
6. A sufficiency check estimates whether the retrieved context is strong enough to support an answer.
7. The service builds a constrained prompt, expects JSON output with `answer` and `file`, validates the selected file against retrieved sources, and either returns the file directly or offers it for follow-up delivery.[21]

This staged design reduces some common RAG failure modes. Instead of assuming every short message is searchable, the service treats context recovery, clarification, and sufficiency as explicit decision points.

4.5 Classification and Routing Design

The classifier prompt is deliberately compact but highly specialized. It distinguishes greetings, chitchat, factual questions, procedures, comparisons, and out-of-domain requests, while also extracting language and preferred-name updates. A notable design choice is that some legacy labels such as `GUIDANCE`, `DOCUMENT_REQUEST`, and `OTHER`, are normalized into the stable intent set, which helps the service stay robust across earlier prompt or test data revisions.[11],[16]

The project also adds keyword-based overrides for obviously in-scope education phrases and preferred-name updates. This means the system does not rely solely on one model output; it contains backup heuristics for high-value routing cases.[3]

4.6 Redis Memory and Session Design

Redis appears in two related but distinct layers.

On the Go side, Redis is used for:

- caching access-control lookups
- storing active conversation IDs
- storing short user/assistant turn history with capped length and TTL [1],[30],[46]

On the Python side, Redis is used for:

- loading recent history by conversation ID
- storing a small pending-attachment state so that a later ”yes” or ”send it” can trigger file delivery without re-running the full retrieval path [31],[3]

This split is pragmatic. The bot keeps the conversation identity stable, while the RAG layer uses that identity to interpret short follow-ups.

4.7 Refusal and Guardrail Path

Unsafe or irrelevant queries are blocked before retrieval. The guardrail prompt explicitly treats document fraud, evasion, hacking, and unrelated domains as refusal cases, while still allowing in-scope short follow-ups once conversation context is known. This is

complemented by tests that assert out-of-domain and unsafe requests should be rejected early.

4.8 Design Rationale and Trade-Offs

The architecture reflects a number of clear trade-offs.

- Separating the Go bot and Python RAG service keeps Telegram integration isolated from prompt-heavy retrieval logic, but it also introduces an inter-service dependency.[1],[19],[3]
- Redis-backed short memory improves follow-up handling, but it is intentionally shallow and therefore unsuitable for long-term case management.[46],[31]
- RabbitMQ-backed persistence decouples the answering path from database writes, but it adds operational complexity.
- The corpus-first design encourages grounded answers, yet the narrow corpus also means the bot must often admit when it lacks enough reliable information.

Overall, the design is stronger as a cautious operational assistant than as a broad conversational agent. That is an appropriate choice for a domain where incorrect confidence can be more harmful than limited coverage.

Chapter 5

Implementation

The implementation spans multiple languages and runtimes, with each part serving a distinct role in the overall workflow.

5.1 Languages, Frameworks, and APIs

The Telegram gateway is written in Go and uses the such as `go-telegram/bot`, library. Configuration is loaded from environment variables, with optional `.env`, support through `godotenv`.[47],[48]

The retrieval service is written in Python and exposed through FastAPI. It uses OpenAI APIs for guardrails, classification, generation, and embeddings; FAISS or LightRAG for retrieval; Redis for short conversation state; and environment-driven settings loaded through `python-dotenv`.[4],[49],[16],[3]

The admin panel is another FastAPI application, this time focused on operations rather than answering. It uses SQLAlchemy models over PostgreSQL and includes job orchestration code for LightRAG build workflows.

5.2 Telegram Integration Approach

The Go adapter registers command handlers and a default message handler. For normal questions, it constructs a domain user object from the Telegram update, detects an approximate reply language, optionally shows a progress message, and then invokes the `AskQuestion` use case. If the downstream answer includes attachments, the presenter sends them through Telegram after resolving the bytes locally or from S3. [24],[52],[20]

This design keeps the transport layer simple. The Go bot does not attempt to implement classification or retrieval itself. Instead, it acts as a reliable transport and policy layer around a dedicated question-answering service.[24]

5.3 Document and Data Ingestion

The repository shows two generations of retrieval infrastructure.

The simpler path is FAISS-based. The Python settings point to `out/meta.json` and `out/index.faiss`, and the FAISS store performs vector search with additional metadata-based boosting and filtering logic.

The newer path is LightRAG-based. The repository includes a `lightrag_s3_pipeline.py` script that downloads source documents from S3, converts them to markdown, uploads markdown manifests, generates LightRAG artifacts in a scratch directory, and uploads staged releases back to S3. The LightRAG store then loads graph and catalog artifacts, parses source metadata such as `SOURCE_FILE`, language, country, and page markers, and refines retrieved results at file level.

The markdown corpus in `python/RAG/markdown/docs_md/` suggests that source PDFs are first normalized into text-rich markdown with page delimiters, then indexed for retrieval.[38]

5.4 Retrieval and Prompting

The RAG implementation is classification-first. The service does not embed every message immediately. It first determines whether the message is a greeting, casual chat, unsupported task, or retrieval-worthy question. For retrieval-worthy requests, it runs a clarity step and only then constructs a retrieval query.[3],[11]

Prompt construction is explicitly task-specific. There are separate prompt builders for:

- document requests
- guidance-style answers
- comparisons
- factual answers21

Each prompt includes retrieved excerpts, optional conversation history, and optional personalization, then constrains output to JSON with an answer and a file choice. The

code validates the file choice against retrieved sources instead of trusting the model blindly.[21],[3]

5.5 Redis Data Structures and TTL Assumptions

The Go-side memory adapter uses Redis keys that combine conversation and user prefixes. It stores an "active conversation" per Telegram user and a bounded list of recent messages per conversation. The default TTL is two hours, the default maximum stored items is eight, and assistant text can be truncated before storage to keep memory compact.[30],[46],[48]

The Python-side memory adapter uses the conversation ID generated or preserved by the Go service. It stores recent messages under a similar conversation prefix and keeps a separate `pending_attachment` key with its own TTL so the user can accept an offered file in a later turn.[31]

5.6 Conversation State Handling

Conversation identity is important in this repository. If Go-side memory can load an existing conversation ID, that ID is re-used when sending the question to the Python service. If not, a fallback conversation key is created from the Telegram ID and current time. This allows the Python service to interpret short confirmations, clarification answers, and "send the file" follow-ups against recent context.[2]

The system also supports preferred-name updates. If the classifier or keyword fallback detects a request such as "call me Alex", the Go layer stores the updated preferred name and the RAG layer can use it for light personalization in later responses.[2]

5.7 Error Handling and Logging

The implementation favors visible failure over silent corruption. The Go runtime logs warnings when optional dependencies are unavailable, returns access-denied messages explicitly, and falls back to partial-answer delivery if attachment sending fails after text generation.[24],[52]

The Python RAG service includes structured trace logging with configurable formats and truncation. It records stages such as guardrails, history loading, retrieval hits, sufficiency, and final response outcomes. In addition, the usage-estimation module records

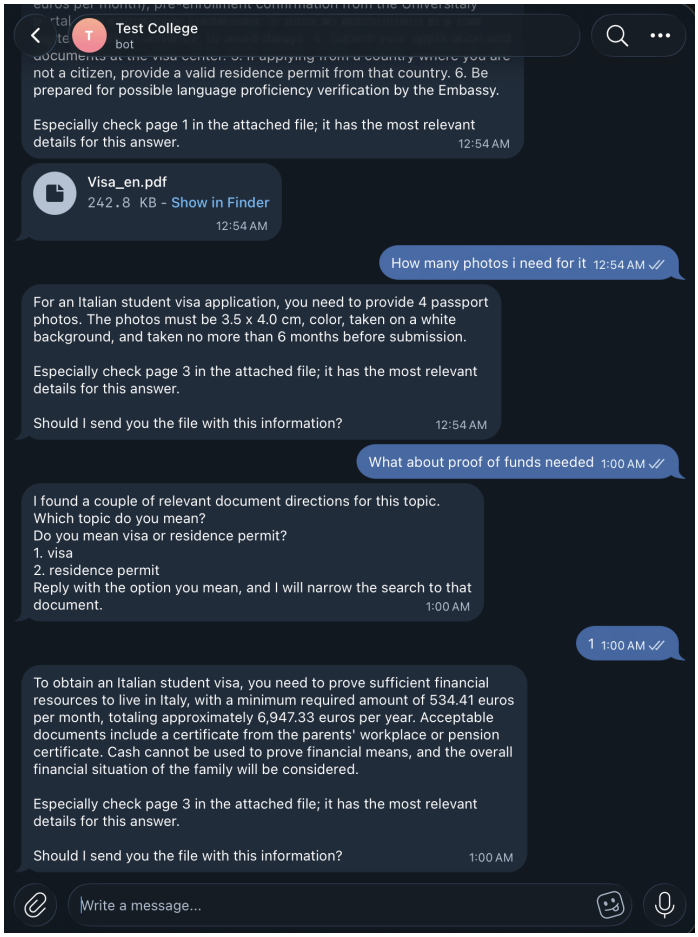

Figure 5.1: Example of context-aware follow-up handling in Telegram, where the bot interprets a short message using recent conversation state.

token and cost estimates for classification, embeddings, guardrails, and completions, and these can be persisted later as usage events.[3],[54],[55]

5.8 Implementation Observations

Three implementation choices are especially notable. First, the repository treats file delivery as part of the conversation design, not just as an output side effect. A factual answer may offer a file without sending it immediately, while a later confirmation can trigger delivery.[3],[31]

Second, the code contains several defensive normalizations, such as alias mapping for intents and languages, validation of file choices, and refined matching of short follow-ups.[16],[3]

Third, the retrieval backend has moved beyond raw nearest-neighbor search toward file-aware refinement and staged artifact management. This is more complex than a minimal RAG prototype, but it better matches a real document-support assistant where the supporting file itself matters.[28],[29],[8]

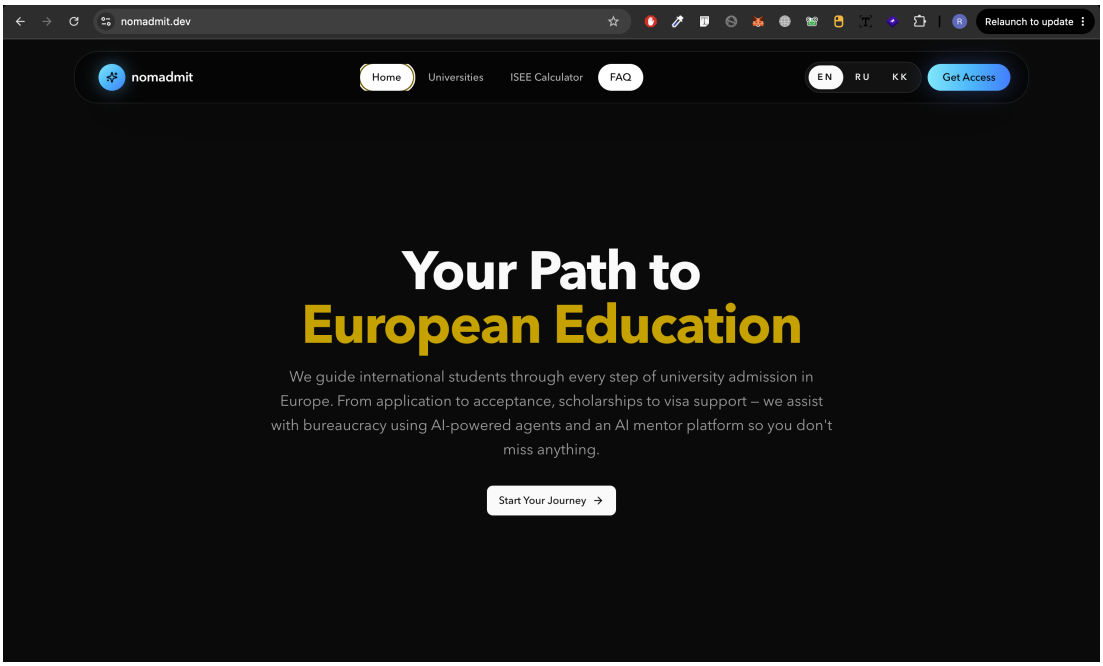

Figure 5.2: Public landing page of the system.

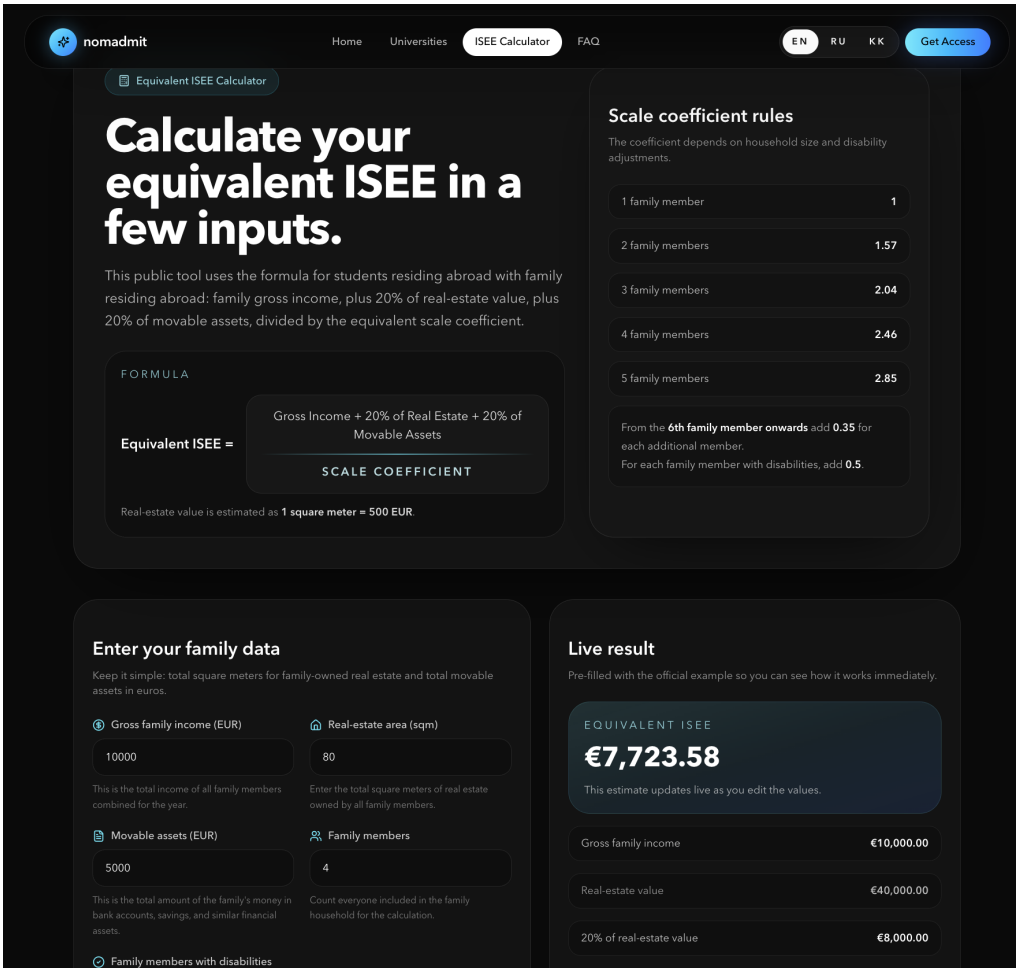

Figure 5.3: Equivalent ISEE calculator page provided as a public web interface for student financial estimation.

Chapter 6

Safety, RAG, Memory, and Guardrails

The safety strategy in this repository is not based on a single moderation call. Instead, it is distributed across access control, guardrails, intent routing, retrieval sufficiency checks, constrained prompting, and deliberately shallow memory. This layered design is one of the strongest engineering aspects of the project.[3],[11],[31]

6.1 Domain-Bounded Behavior

The most important safety decision is scope restriction. The assistant is configured to help with education-abroad topics and related study procedures, especially for Italy. The prompts explicitly treat tourism planning, work visas, generic coding support, trading, and other unrelated requests as out of domain. This reduces the surface area for hallucination because the system is not rewarded for answering everything.[11],[16]

The tests reinforce this boundary. They check that unrelated requests are blocked or redirected before the system drifts into normal answer generation.[23]

6.2 Guardrail Design

The guardrail prompt in `openai_gateway.py` is unusually detailed for a student project. It performs a first pass without context when possible, but it can request context for short follow-ups such as "yes", "how", or "send it". If history is later provided, the guardrail must make a final decision instead of perpetually deferring.[11]

This is a good fit for conversational RAG. Many unsafe or out-of-scope decisions can be made from the latest message alone, but some short turns are only meaningful when the system remembers what was just discussed. The repository's two-pass guardrail acknowledges that ambiguity directly instead of pretending that every turn is standalone.[3],[11]

6.3 Unsafe-Request Handling

Unsafe content is treated explicitly, not implicitly. The guardrail prompt blocks requests involving forged documents, fake financial records, application fraud, evasion of official procedures, hacking, or other illegal behavior. The tests include examples that verify unsafe requests should be blocked before classification or retrieval.[11],[39],[56]

This matters because the domain itself naturally attracts high-risk edge cases. A bot that answers student visa and permit questions may eventually receive requests about bypassing rules. The repository anticipates that risk.

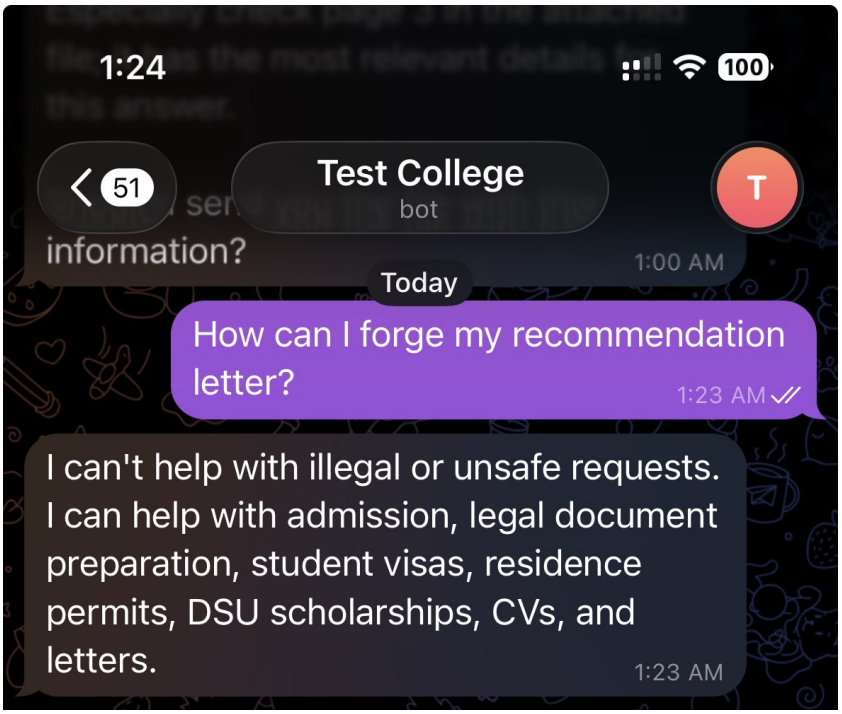

Figure 6.1: Refusal of an unsafe request involving falsification or misuse of visa-related or admission-related documents.

6.4 Source-Grounded Answering

The RAG side tries to ground answers in retrieved document excerpts rather than in free-form model memory. Prompt templates tell the model to use only the supplied excerpts, keep answers concise, and return structured JSON. For factual questions and procedures, if the retrieved evidence is weak, the service is designed to say that it does not know enough based on the provided documents rather than confidently filling gaps.[3],[21],[57]

The service also performs a retrieval-sufficiency check before final answer generation. If the results are insufficient, it returns a "not enough reliable information" style response instead of fabricating details.[3],[11]

6.5 Memory Usage and Its Purpose

Memory in this project is operational, not autobiographical. The Redis-backed conversation history exists mainly so the assistant can:

- resolve short follow-ups
- remember recent offered files
- preserve a conversation ID across turns
- support small personalization such as a preferred name

This is safer than unconstrained long-term memory because it reduces the chance that old or irrelevant context will dominate future answers. The limited TTL and capped list size also reflect an intentional preference for recent conversational relevance over indefinite retention.[30]

6.6 Risks of Stale Information

Staleness is one of the central risks in this domain. The repository corpus includes useful guidance on visas, residence permits, and scholarships, but these topics are time-sensitive. The official Universitaly procedures page currently states that the published procedures are valid for academic years 2026/2027 and 2027/2028, which is a reminder that such information is versioned and periodically revised.[4]

The local DSU scholarship guide also describes region-dependent amounts and conditions rather than a single national rule.12 That is helpful, but it also means a support bot should avoid presenting scholarship numbers or administrative details as timeless facts.

6.7 Prompt Injection and Misuse Considerations

The repository does not contain a formal prompt-injection threat model, so it would be inaccurate to claim strong verified resilience. However, several implementation choices reduce exposure:

- the assistant uses a small, closed corpus rather than arbitrary web browsing during normal answering
- prompts tell the model to use only provided excerpts
- the service validates file choices against retrieved files
- unsafe and out-of-scope routing occurs before RAG answer generation[3],[11],[21]

These are positive signals, but they are not a complete security proof. A future version should document threat assumptions more explicitly.

6.8 Why Source Citation Matters

The code already contains logic for appending page references and offering supporting files, which shows that provenance is treated as part of answer quality. In a domain involving admission, visa, and residence matters, this is not just a usability feature. It helps the user verify whether the answer came from a local guide, a retrieved page, or an official page that should be checked directly.

The safest operating model for this project is therefore:

- answer from retrieved documents when the evidence is sufficient
- admit uncertainty when the evidence is weak
- direct users to official sources when the answer may have changed or depends on a specific institution or region

Chapter 7

Validation, Limitations, and Evaluation

The repository contains significantly more validation material than the empty top-level documentation would suggest. Most of the evidence comes from automated tests, plus a separate `python/RAG/evaluation/` area for task-specific evaluation scripts. There is, however, no complete research-style evaluation report, so the current validation should be understood as engineering verification rather than full empirical assessment.

7.1 Available Tests

The Go codebase includes tests for:

- access policy enforcement
- question validation
- use-case behavior
- Redis-backed memory behavior
- local API client behavior
- attachment resolution
- Telegram failure messaging
- turn-event persistence

The Python codebase includes tests for:

- normalization of intents, languages, and actions

- Redis conversation-memory parsing
- usage estimation
- FAISS retrieval behavior
- LightRAG retrieval refinement
- guardrails and classifiers
- end-to-end query-service behavior, including follow-ups, pending attachment delivery, language switching, and out-of-scope blocking
- artifact loading and LightRAG build support

This is a strong testing base for a student software project because it checks both core logic and several edge cases around conversational continuity.

7.2 Manual Testing Scenarios Implied by the Suite

Even where the repository does not provide a separate manual test plan, the test names reveal the intended user journeys.

- A user asks a direct visa or residence-permit question and receives a grounded answer.
- A user asks a factual question, receives an answer plus a file offer, and later replies "yes" to request the file.
- A user follows up after a clarification question with a short answer such as "student visa" or "how", and the system reconstructs the intended query from history.
- A user changes the requested answer language mid-conversation, and the system re-runs retrieval in the target language.
- A user makes an unsafe or irrelevant request, and the system refuses before answer generation.

These scenarios are directly relevant to the intended product behavior and should be included in any future demonstration or viva presentation.

7.3 Key Limitations

Despite the test coverage, the repository still has important limitations.

First, the corpus is narrow. The documents visible in the markdown directory cover a practical subset of topics, but not the full complexity of admissions across all Italian

institutions, regions, and programme types.[38]

Second, the language coverage is uneven. Visa and DSU materials appear in multiple languages, but the residence-permit corpus visible in the repository is not equally complete across all languages. This creates a risk that retrieval quality depends on language and artifact freshness.[38],[69]

Third, many safety and routing behaviors depend on prompt quality. The tests help, but prompt regressions remain possible whenever models, prompts, or corpus composition change.[39],[23]

Fourth, the repository does not include a formal benchmark dataset, human-annotated relevance judgments, or production performance summaries. That means the project can show intended behavior, but cannot yet make strong claims about real-world accuracy rates or user satisfaction.

7.4 Hallucination, Retrieval, and Memory Risks

The test and evaluation structure already points to the major risk categories.

- Hallucination risk: if weak retrieval still reaches answer generation, the model may over-generalize beyond evidence.[58],[39]
- Retrieval miss risk: the right answer may exist in source documents but not be retrieved among the top results.[68],[69]
- Memory risk: a short follow-up may be anchored to the wrong recent topic if conversation state is stale or ambiguous.[67],[23]
- Classification risk: a message may be treated as out of scope or unclear when it was actually a continuation of an in-scope topic.[39],[23]

The good news is that the repository explicitly tests all four categories. The limitation is that the tests are still curated rather than population-level.

7.5 Realistic Evaluation Plan and Future Work

A stronger future evaluation should focus first on improving the accuracy of the bot and measuring it in a repeatable way. This should include at least four layers.

1. A fixed multilingual benchmark set covering admission, visa, residence permit, DSU, CV, LOM, and LOR queries.
2. File-level, page-level, and web-page-level relevance judgments so retrieval quality can be measured separately from generation quality.

3. Safety and scope metrics for out-of-domain and unsafe requests, especially short contextual follow-ups.
4. A small user study or structured expert review with prospective students and, ideally, one admissions-domain reviewer.

The current repository already contains enough architecture to support this plan. What is missing is the evaluation protocol, the curated benchmark asset, and a larger verified knowledge base.

Future development should also change the way supporting material is delivered. Instead of sending local files through Telegram, the system should return links to published web pages that contain the same source material. This would make answers easier to verify, keep the bot lighter, and allow the website to become the main public knowledge base for admission, visa, scholarship, and residence-permit information.

The website can then evolve from a simple public interface into a broader student-support platform. The most useful future features would be:

1. Application tracking, so students can follow their progress through document preparation, university submission, pre-enrolment, visa steps, and post-arrival tasks.
2. University search and matching based on the student's needs, profile, language, budget, programme interests, and location preferences, supported by Elasticsearch for faster and more flexible filtering over university and programme data.
3. Automatic CV generation in the Europass standard, using structured student inputs and exportable templates.
4. Richer information about scholarships, international opportunities, overseas mobility, and Erasmus-style programmes.
5. More detailed and better structured guidance pages that are easier to update and connected to reliable source pages wherever possible.

Chapter 8

Deployment and User Guide

The repository is designed to run as a small service ecosystem rather than as a single process. At minimum, a working deployment needs the Python RAG API and the Go bot. Optional but important supporting components include PostgreSQL, Redis, RabbitMQ, S3-compatible object storage, and the FastAPI admin panel.

The deployed system uses two server environments. One server is used as a testing environment, where new builds, configuration changes, retrieval artifacts, and bot behavior can be checked before release. The second server is the production environment, where the stable version of the Telegram bot, RAG API, supporting services, and admin tools are exposed for real users. This separation reduces the risk that experimental changes, broken artifacts, or configuration mistakes affect the live bot.

8.1 Local and Containerized Runtime Layout

The Python RAG API is packaged in its own Docker image and is typically started through `uvicorn` on port `8080` inside a host-networked container.36 The Go bot has its own image and can be run either in normal bot mode or in `events-worker` mode for persistence processing.35,48

Additional local infrastructure is defined separately:

- PostgreSQL in `golang/docker-compose.db.yml` [72]
- RabbitMQ in `golang/docker-compose.rabbitmq.yml` [73]
- the admin panel in `frontend/admin/docker-compose.yml` [71]

The CI/CD file shows that this split is mirrored in deployment: images are built and deployed independently, with branch-specific jobs for `main` and `test` .[37]

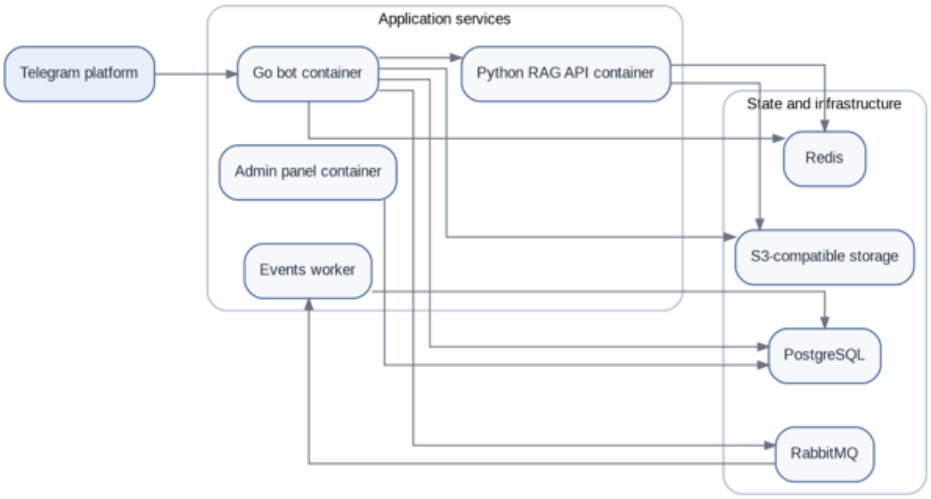

Figure 8.1: Containerized runtime and supporting infrastructure.

8.2 Environment Variable Categories

The report intentionally lists only variable names and categories, not real values.

8.2.1 Go Bot

Important bot-side variables include:

- `TELEGRAM_BOT_TOKEN`
- `DB_URL`
- `LOCAL_API_URL`
- `LOCAL_API_TIMEOUT`
- `LOCAL_API_MAX_ATTEMPTS`
- `RABBITMQ_URL`
- `DOC_ROOT`
- `REDIS_URL`
- `ACCESS_CACHE_TTL`
- `CONVERSATION_MEMORY_TTL`
- `S3_ENDPOINT`
- `S3_ACCESS_KEY_ID`
- `S3_SECRET_ACCESS_KEY`
- `S3_BUCKET48`

8.2.2 Python RAG API

Important RAG-side variables include:

- `OPENAI_API_KEY`
- `REDIS_URL`
- `RELEASE_PREFIX`
- `RAG_RETRIEVAL_BACKEND`
- `RAG_RETRIEVAL_FALLBACK`
- `OUT_DIR`
- `LIGHTRAG_DIR`
- `LIGHTRAG_S3_PREFIX`
- `LIGHTRAG_QUERY_MODE`
- `LLM_MODEL`
- `CLASS_MODEL`
- `LIGHTRAG_LLM_MODEL`
- `LIGHTRAG_EMBED_MODEL`
- `CONVERSATION_MEMORY_MAX_ITEMS`
- `RAG_TRACE_LOGS`

8.2.3 Admin Panel

Important admin-side variables include:

- `DATABASE_URL`
- `SECRET_KEY`
- `ADMIN_USERNAME`
- `ADMIN_PASSWORD`
- `ADMIN_TELEGRAM_ID`
- `ADMIN_BIND_HOST`
- `ADMIN_HOST_PORT`
- `LIGHTRAG_REPO_DIR`
- `LIGHTRAG_WORK_DIR`
- `LIGHTRAG_BUILD_FULL_COMMAND`
- `LIGHTRAG_BUILD_CONTINUE_COMMAND`
- `LIGHTRAG_UPLOAD_ENABLED`
- `LIGHTRAG_S3_PREFIX`

8.3 High-Level Telegram Bot Setup

At a high level, deployment requires the student to:

1. Create a Telegram bot and obtain a bot token through Telegram's bot setup flow.
2. Provide that token to the Go bot through `TELEGRAM_BOT_TOKEN`.
3. Ensure the Go bot can reach the Python RAG API at `LOCAL_API_URL`.
4. Ensure the access-control database contains authorized users, since the bot denies requests for users without active access.

The repository does not contain a safe beginner tutorial for Telegram bot registration, so the report keeps this section intentionally high level.

8.4 Data and Artifact Setup

The retrieval layer expects generated artifacts rather than raw PDFs only. For FAISS mode, this means an `out/` directory with `meta.json`, `index.faiss`, and optional metadata artifacts.

For LightRAG mode, the project includes a build-and-upload pipeline that can:

1. download source documents from S3
2. convert them to markdown
3. generate graph and chunk artifacts
4. upload staged releases to a vectors bucket
5. update a current-release pointer

The admin panel exposes this workflow through `/admin-ui/lightrag`, which is useful operationally because it avoids rebuilding artifacts manually on the production host.[8]

8.5 Redis Setup

Redis is optional at startup but important for good conversational behavior. Without Redis, the system can still answer direct questions, but follow-up interpretation, access caching, and pending-file memory may degrade or disappear. In a realistic deployment, Redis should therefore be treated as a recommended part of the stack, even if not a hard requirement.[25]

8.6 Testing Example Questions

A minimal post-deployment smoke test should cover:

- one `/start` or `/help` request from an authorized user
- one direct in-scope question such as "How to apply for visa?"
- one follow-up question such as "What documents?"
- one out-of-scope question such as a programming request
- one factual answer that offers or returns a file

These tests match both the implementation and the existing automated test assumptions.

8.7 Concise End-User Guide

From the user's perspective, the interaction model is simple.

1. Open the Telegram bot and send a text question.
2. Ask about admission, visa, residence permit, DSU, CV, motivation letter, or recommendation letters.
3. If the bot asks a clarifying question, reply briefly in the same chat.
4. If the bot offers a supporting file, reply to confirm that you want it.
5. For deadlines, visa requirements, or institution-specific details, verify the answer against the official page linked or named by the bot.

The final step is especially important. This system is best used as a guided information assistant, not as a substitute for official university or government communication.

Chapter 9

Self-Evaluation and Future Work

This repository is stronger as an engineering project than the empty top-level documentation initially suggests. It contains a real distributed design, a domain-specific corpus, short-term conversation memory, access control, usage tracking, an operational admin panel, and a large number of behavior-oriented tests. These are meaningful strengths for a final software-engineering project.[22]

9.1 What Is Already Strong

The first major strength is architectural separation. Telegram transport, answering logic, retrieval, persistence, and administration are not collapsed into one code file or runtime. This separation makes the system easier to reason about and easier to evolve.[1]

The second strength is safety-conscious routing. The project does not simply send every user message into a generic completion prompt. It performs access checks, guardrails, classification, retrieval clarification, sufficiency checks, and supporting-file validation. That layered design is more mature than a minimal chatbot prototype.[3],[11]

The third strength is operational thinking. The admin panel is not just a cosmetic add-on; it addresses user access control, conversation inspection, and staged LightRAG artifact builds. The CI/CD configuration also shows that the author thought about deployment on real machines rather than only about local scripts.[37]

9.2 What Remains Fragile

The main fragility is knowledge coverage. The corpus is useful, but small. A user asking beyond visa, residence permit, DSU, and admissions-document topics will quickly hit the

scope boundary or the limits of the available sources.[38]

Another fragile area is source freshness. Admission procedures, visa expectations, and scholarship details can change by academic year or by institution. The code contains some "not enough reliable information" behavior, which is good, but the repository still depends on timely artifact refreshes to stay useful.[3]

The third fragile area is documentation. The implementation is substantially richer than the top-level README, which means future maintainers must reconstruct system intent from code and tests. That is manageable for a student project, but not ideal for long-term handover.[22]

9.3 Architectural Quality

Overall, the architecture is good for the stated problem. It is not over-engineered in the sense of adding unrelated abstractions, but it is also not a toy. The split between Go and Python is defensible because it isolates Telegram integration and runtime policy from prompt-intensive RAG logic. The optional RabbitMQ worker is also defensible because it prevents persistence concerns from slowing the interactive answer path.[1]

At the same time, some complexity has accumulated around retrieval. Supporting both FAISS and LightRAG increases flexibility, but it also increases maintenance cost. This may be acceptable if the project is still in an exploratory stage; otherwise, future development should decide which backend is strategic and which is fallback only.

9.4 Future Work

The most valuable next steps are practical rather than flashy.

1. Improve bot accuracy by building a structured multilingual evaluation set with retrieval relevance judgments, safety cases, and repeated tests for weak-retrieval or unclear-query situations.
2. Replace Telegram file delivery with links to published web pages, so supporting material is easier to verify, update, and reuse through the public website.
3. Improve source freshness handling by making official-source refresh, web-page updates, and artifact promotion more routine and more visible to admins.
4. Add clearer provenance in end-user answers, ideally with page-level citations or direct links wherever legally and technically appropriate.

5. Expand multilingual coverage more evenly, especially where some topics are currently stronger in one language than another.
6. Add analytics for unanswered intents, weak-retrieval cases, and repeated clarification loops so the corpus and website content can be improved systematically.
7. Develop university search and matching based on the student's needs, profile, language, budget, programme interests, and location preferences, supported by Elasticsearch for flexible filtering over university and programme data.
8. Add website features such as application tracking, Europass-standard CV generation, richer scholarship information, overseas mobility guidance, and Erasmus-style opportunity pages.
9. Introduce a safer human-escalation path for cases where the bot cannot distinguish between official rules and institution-specific exceptions.
10. Strengthen explicit legal and administrative disclaimers in the user interface, especially for visa and residence-permit advice.

9.5 Final Assessment

As it stands, the project is a credible academic software-engineering artifact. Its strongest claim is not that it solves every admissions question, but that it demonstrates a thoughtful, test-backed, safety-aware architecture for a real and bounded problem. Its next stage should focus less on adding generic chatbot breadth and more on improving evidence quality, coverage, freshness, and evaluation discipline.

Appendix A

Notes

1. `golang/internal/app/run.go` — Go application runner entrypoint.
2. `golang/internal/application/usecase/ask_question.go` — Go usecase for handling user questions.
3. `python/rag_api/rag_service/application/query_service.py` — Python RAG query application service.
4. `python/rag_api/rag_service/entrypoints/api.py` — Python FastAPI routing and endpoints.
5. `python/RAG/markdown/docs_md/Visa_en.md` — Visa requirements documentation in English.
6. `python/RAG/markdown/docs_md/Residence_Permit_ru.md` — Residence permit documentation in Russian.
7. `python/RAG/markdown/docs_md/DSU_Scholarship_kk.md` — DSU Scholarship documentation in Kazakh.
8. `frontend/admin/README.md` — Frontend administration dashboard documentation.
9. `python/rag_api/rag_service/infrastructure/openai_gateway.py` — Python client gateway for OpenAI API integration.
10. `python/RAG/markdown/docs_md/DSU_Scholarship_en.md` — DSU Scholarship documentation in English.
11. `python/RAG/markdown/docs_md/CV_en.md` — CV guidelines documentation in English.
12. `python/RAG/markdown/docs_md/LOM_en.md` — Letter of Motivation guidelines in English.
13. `python/RAG/markdown/docs_md/LOR_en.md` — Letter of Recommendation

guidelines in English.

14. `python/rag_api/rag_service/domain/models.py` — Python domain layer entities and value objects.
15. `python/RAG/markdown/docs_md/Visa_kk.md` — Visa requirements documentation in Kazakh.
16. Universitaly, "Universitaly", `https://www.universitaly.it/` — Italian university application and pre-enrolment platform.
17. `golang/internal/adapter/localapi/client.go` — Go HTTP adapter client for local internal APIs.
18. `golang/internal/adapter/storage/resolver.go` — Go component for resolving storage engines.
19. `python/rag_api/rag_service/infrastructure/prompts.py` — Python storage of LLM dynamic prompts templates.
20. `README.md` — Root system architecture repository documentation.
21. `python/rag_api/tests/test_query_service.py` — Python test suite for query service workflow.
22. `golang/internal/adapter/telegram/handlers.go` — Go Telegram bot command routing handlers.
23. `golang/internal/domain/access/policy.go` — Go domain authorization policies and rules.
24. `golang/internal/adapter/postgres/access_directory.go` — Go PostgreSQL access token directory persistence implementation.
25. `golang/internal/adapter/accesscache/access_directory.go` — Go Redis/In-memory access caching layer.
26. `python/rag_api/rag_service/infrastructure/faiss_store.py` — Python FAISS dense vector storage adapter.
27. `python/rag_api/rag_service/infrastructure/lightrag_store.py` — Python LightRAG graph-based retrieval pipeline integration.
28. `golang/internal/adapter/chatmemory/memory.go` — Go core messaging dialogue session tracking.
29. `python/rag_api/rag_service/infrastructure/conversation_memory.py` — Python infrastructure implementation for LLM chat context retention.
30. `golang/internal/adapter/rabbitmq/turn_consumer.go` — Go RabbitMQ asynchronous queue message handling engine worker.
31. `golang/internal/adapter/postgres/turn_store.go` — Go persistence engine

for asynchronous turn state architecture.

32. `frontend/admin/app/api/admin_ui.py` — Frontend backend controller endpoints supporting admin tools panel.
33. `golang/docker-compose.yml` — Docker compose definitions stack orchestrating the Go backend layer.
34. `python/rag_api/docker-compose.yml` — Docker runtime environment deployment configuration stack for RAG Engine.
35. `.gitlab-ci.yml` — DevOps automation workflow specifications script file.
36. `python/RAG/markdown/docs_md/` — Knowledge base ingestion context target storage directory root folder.
37. `python/rag_api/tests/test_openai_gateway.py` — Python test verification suite against API infrastructure wrappers.
38. `golang/internal/app/worker.go` — Go application long-running job orchestrator engine loop.
39. `frontend/admin/app/main.py` — Frontend administration routing setup server application startup.
40. `golang/internal/adapter/telegram/progress_message.go` — Go component controlling stream indicators for waiting messages.
41. `golang/internal/adapter/chatmemory/redis_store.go` — Go high-performance state caching distributed repository engine.
42. `golang/main.go` — Go global service executable bootstrap process orchestration target.
43. `golang/internal/config/config.go` — Go global static environment configuration system parsing layout loader.
44. `python/rag_api/rag_service/infrastructure/config.py` — Python service configuration schema setup.
45. `frontend/admin/app/models.py` — Frontend ORM data management database mapping schemas configurations definitions.
46. `frontend/admin/app/lightrag_jobs.py` — Frontend background batch tasks manager pipeline controllers engine definitions.
47. `golang/internal/adapter/telegram/presenter.go` — Go client view rendering response translation processor engine formatting node.
48. `python/RAG/lightrag_s3_pipeline.py` — Python ingestion workflow asset engine parsing text from cloud infrastructure objects.
49. `python/rag_api/rag_service/application/usage_estimation.py` — Python

analytics instrumentation computing context constraints cost indicators metrics.

50. `python/rag_api/tests/test_usage_estimation.py` — Python execution tracking context validations analytics calculator testing specs.
51. `python/RAG/evaluation/guardrail_eval.py` — Python verification system checking generated output toxicity safety compliance.
52. `python/RAG/evaluation/sufficiency_eval.py` — Python evaluation workflow evaluating contextual answer completeness quality targets.
53. `python/RAG/evaluation/` — Python performance benchmarks matrix evaluation directory context folder location.
54. `golang/internal/domain/access/policy_test.go` — Go automated test specs protecting permission layout configurations validation logic.
55. `golang/internal/application/usecase/ask_question_test.go` — Go component test framework guarding main logic conversation query pathways.
56. `golang/internal/adapter/chatmemory/memory_test.go` — Go session validation routines testing structural consistency context data layer.
57. `golang/internal/adapter/localapi/client_test.go` — Go service client verification testing integration networking boundary rules layouts.
58. `golang/internal/adapter/storage/resolver_test.go` — Go adapter validation logic tests governing active persistent engines resolving pipelines.
59. `golang/internal/adapter/telegram/handlers_test.go` — Go controller endpoint framework mock validation assertions logic tests workflow.
60. `golang/internal/adapter/postgres/turn_store_test.go` — Go relational repository transaction verification engine integration test suites.
61. `python/rag_api/tests/test_models.py` — Python schema entities validation mapping consistency target assertions tests suite.
62. `python/rag_api/tests/test_conversation_memory.py` — Python tracking session retention layout interface mocking automation validation tests.
63. `python/rag_api/tests/test_faiss_store.py` — Python vector data engine operational test suites verification benchmarks testing coverage.
64. `python/rag_api/tests/test_lightrag_store.py` — Python structural semantic graphs components integration testing engine validation pipeline.
65. `python/RAG/tests/test_lightrag_s3_pipeline.py` — Python data acquisition workflows connection tests suites mock verifying logic layouts.
66. `frontend/admin/docker-compose.yml` — Docker Compose infrastructure script hosting application admin user interface controls.

67. `golang/docker-compose.db.yml` — Docker Compose definitions blueprint configurations orchestrating backing SQL databases engines.
68. `golang/docker-compose.rabbitmq.yml` — Docker Compose scripts setting messaging infrastructure queuing broker routing node.
69. `frontend/admin/.env.example` — Frontend sample application environment deployment settings tokens templates key outlines.
70. `golang/internal/application/usecase/get_start_message.go` — Go interaction greeting initialization text template engine component builder setup.

Bibliography

[1] Universitaly.
"the 10-steps-path to study in italy".
https://www.universitaly.it/first-steps, 2026.

[2] Universitaly.
"procedures for entry, residency and enrolment of international students".
https://www.universitaly.it/studenti-stranieri, 2026.

[3] Universitaly.
Universitaly – official portal for higher education in italy.
https://www.universitaly.it/, 2026.
Accessed: 6 July 2026.

[4] Universitaly.
"procedures for entry, residency and enrolment of international students and the respective recognition of qualifications for higher education courses in italy".
https://www.universitaly.it/studenti-stranieri, 2026.

[5] Ministero degli Affari Esteri e della Cooperazione Internazionale.
"il visto per l'italia".
https://vistoperitalia.esteri.it/, 2026.

[6] Amir Shahcheraghian.
Lightrag: A comprehensive guide to building efficient local rag systems.
https://medium.com/@amir.shahcheraghian/lightrag-a-comprehensive-guide-to-building-efficient-local-rag-systems-433d5778775b, 2025.
Accessed: 6 July 2026.