Global Real-Time Translator Chat - Requirements Doc

1. Introduction

The Global Real-Time Translator Chat is an interactive chat application that enables users to communicate in real time across language barriers. Users send messages in their preferred input language, and recipients receive automatic translations in their chosen output languages.

2. Project Goals

- Facilitate seamless multilingual communication. - Demonstrate full-stack expertise with real-time systems and AI integration. - Provide a polished, deployable demo for portfolio and interviews.

3. Functional Requirements

3.1 User Management: - Sign up / Login (email, OAuth). - User profile with default input/output languages. 3.2 Chat Rooms: - Create/join public and private rooms. - Direct one-to-one chat support. 3.3 Messaging: - Real-time text messaging via WebSockets. - Display original and translated text. 3.4 Translation Service: - Auto-detect source language. - Translate to each recipient's preferred language. - Cache frequent translations. 3.5 Message History: - Store messages and translations in database. - Paginated retrieval for chat history. 3.6 Presence & Typing Indicators: - Show online/offline status. - Typing notifications.

4. Non-Functional Requirements

- Latency: <200ms end-to-end translation display. - Scalability: Support 10,000+ concurrent users. - Reliability: 99.9% uptime SLA. - Security: TLS, JWT auth, input validation. - Observability: Metrics, logging, error tracking.

5. User Stories

- As a user, I want to choose my input and output languages so I can chat comfortably. - As a user, I want to see my message in original and translated form immediately. - As a user, I want to scroll back to view past conversations with translations.

6. Data Model

Entities: - User(id, name, email, password_hash, default_input_lang, default_output_lang) - Room(id, name, is_private, created_by) - RoomMember(room_id, user_id, joined_at) - Message(id, room_id, sender_id, text_original, lang_original, timestamp) - Translation(message_id, target_lang, text_translated, provider, latency_ms)

7. API Endpoints

GET /api/rooms - List rooms POST /api/rooms - Create room GET /api/rooms/{roomId}/messages?page=N - Get messages WebSocket: connect /rooms/{roomId} - Join room WebSocket: NEW_MESSAGE event - Send message WebSocket: TRANSLATED_MESSAGE event - Receive translation

8. System Architecture

Clients ↔ API Gateway ↔ Chat Service ↔ Redis Pub/Sub ↔ Translation Service ↔ Database Use WebSockets for real-time transport, Redis for message fan-out and caching.

9. Tech Stack

- Frontend: Next.js, React, Tailwind CSS, Socket.IO - Backend: Node.js, TypeScript, Express/tRPC - Database: PostgreSQL, Prisma ORM - Cache & Pub/Sub: Redis - Translation: OpenAI GPT-4o / DeepL API, local LLM fallback - Deployment: Vercel, Kubernetes (Cloud Run), Docker - CI/CD: GitHub Actions - Monitoring: Prometheus, Grafana, Sentry

10. Timeline (3 hr/day)

Week 1: ERD, API spec, scaffold repo, auth setup Week 2: WebSocket chat, message persistence, client UI Week 3: Translation service integration, caching Week 4: Fan-out translation, storage, UI rendering Week 5: History pagination, notifications, security Week 6: Admin features, testing, CI/CD, deployment