NIT3004 – IT Capstone Project 2

Semester 2, Block 4 (H2B4) – Group 14

Technical Implementation Document: TranslateAI

Student Name and ID: Nachiket Patel (s8082887)

Date of Submission: 16th November 2025

Lecturer: Dr Gongqi (Joseph) Lin

Unit Convenor: Csaba Veres

Table of Contents:

1. Introduction

1.1 Purpose of the Document

This Technical Implementation Document provides an elaborative breakdown of the design choices and the programming decisions taken for the creation and development of TranslateAI. It is a multilingual, AI-powered web application designed to provide real-time translation across different input modes. The project was developed for NIT3004 – IT Capstone 2 at Victoria University, portraying the integration of artificial intelligence and modern web technologies.

The purpose of this document is to explain the complete technical implementation of the system, starting from the initial setup to the final deployment. It will cover every core component including the frontend, backend, database, and security subsystems.

The document will also provide future developers and assessors with sufficient technical information to reproduce the project in a similar environment. It contains discussions of code snippets, listings and locations of key files, explanations of data flows, and the configuration details for secure operations.

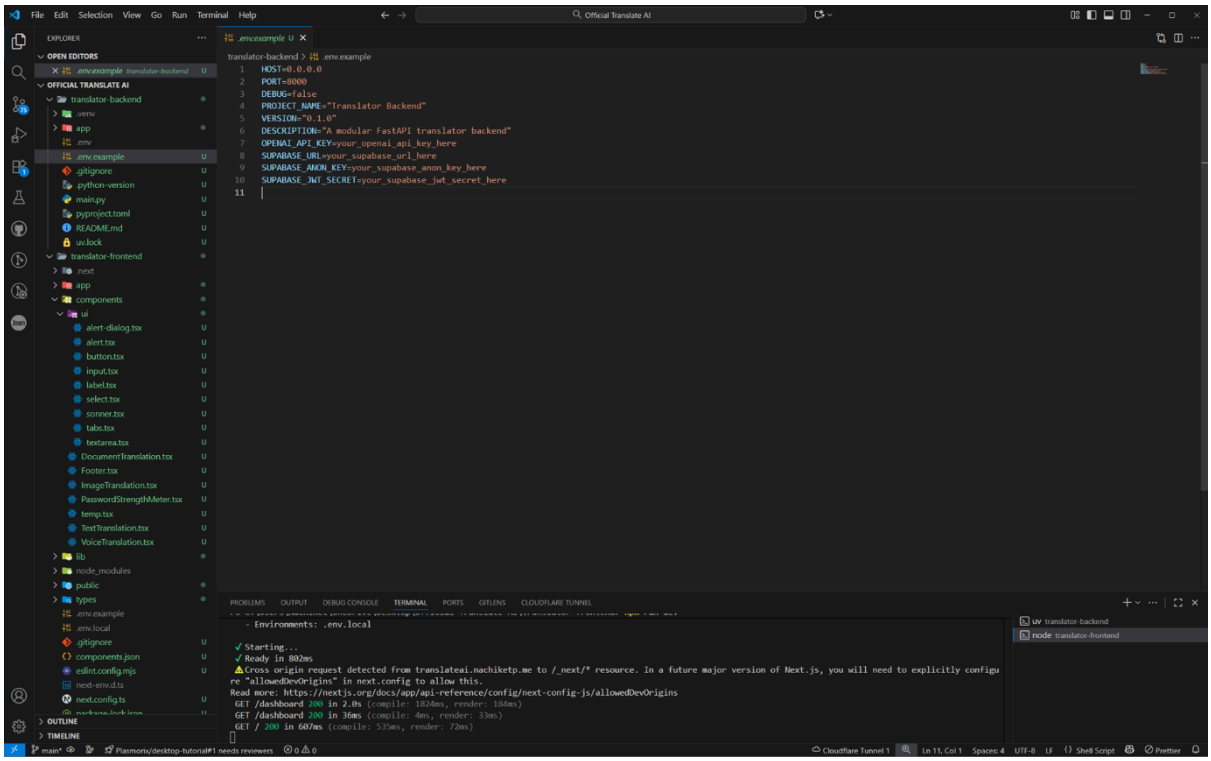


Figure 1. VS Code workspace displaying both the "translator-backend" and "/translator-frontend" folders

1.2 Overview of the Project

Originally, TranslateAI began as a group project under the supervision of Dr Gongqi (Joseph) Lin and Dr. Chenhao Xu during IT Capstone 1 but was completed independently, beginning from IT Capstone 2 by Nachiket Patel (s8082887) after the original group members did not show any effort to be proactive and collaborative. Over four nights of dedicated and focused development (26 – 29 October 2025), the entire system was designed, implemented, and tested from scratch, integrating both frontend and backend components into a fully functional, real-time translation platform.

TranslateAI provides users with an intuitive browser-based environment to translate text, voice, images, and documents into any target language. Unlike existing translation tools which separate these capabilities and restrict real-time translation to mobile devices (GoogleTranslate), TranslateAI delivers an integrated, cross-platform experience accessible from both desktop and mobile browsers.

The system architecture employs the following:

- Next.js 16 and TailwindCSS for the minimal, yet reactive frontend.
- FastAPI and LangChain for managing translation workflows and OpenAI API calls.
- Supabase (specifically PostgreSQL) for secure user management and translation history
- Cloudflare Tunnel and DNS for an encrypted global deployment.

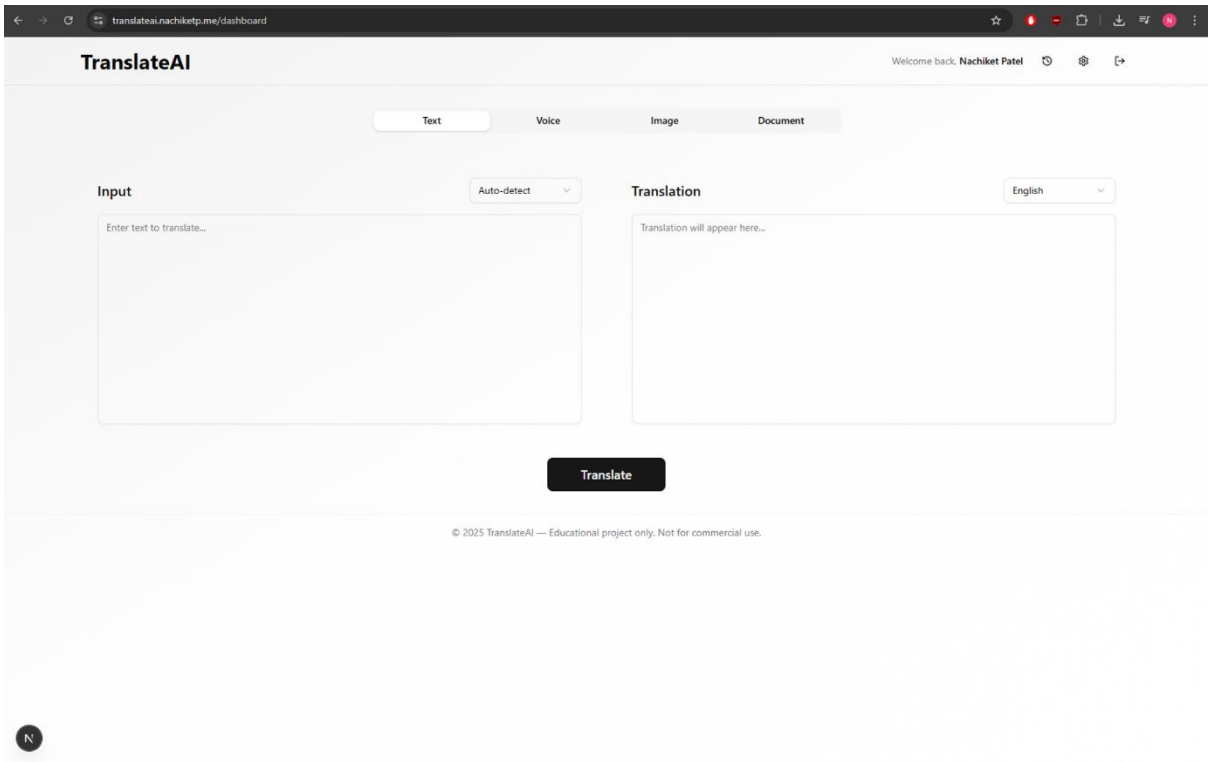


Figure 2. TranslateAI landing page with active translation user interface.

1.3 Functional Requirements and Use Cases

Use Case ID	Description	Input	Expected Output
Use Case 1	User registration/login via Supabase Authenticator system (Email/Password or OAuth via Google/GitHub).	Credentials	Authenticated session token
Use Case 2	Text-to-text translation	Text and Target Language	Translated text displayed instantly
Use Case 3	Voice-to-text translation	Audio file and Target Language	Extracted text and Translated text
Use Case 4	Image-to-text translation	Image file and Target Language	Extracted text and translated text
Use Case 5	Document-to-text translation	Document and Target Language	Extracted text and translated text
Use Case 6	View/Delete Translation History	User ID	Chronological translation records from Supabase

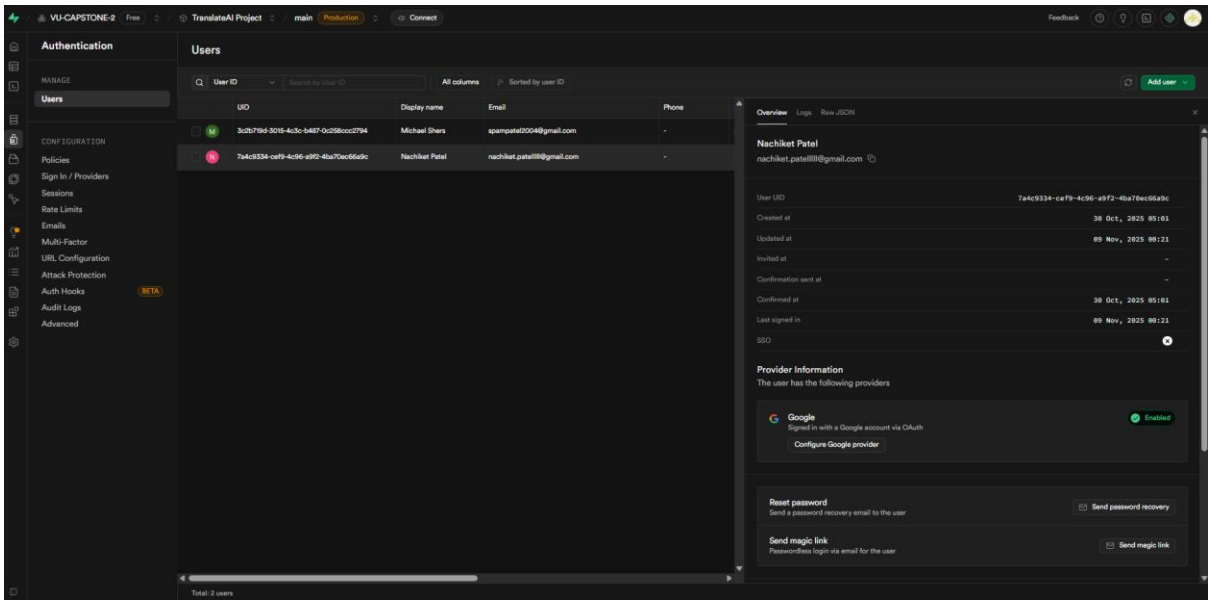


Figure 3. Supabase dashboard: Authentication tab showing user entries and OAuth provider.

1.4 System Overview

TranslateAI utilises a three-tier client-server architecture secured through Cloudflare:

- Frontend (Presentation Layer) – Handles UI rendering, form submission, and user authentication through Supabase.
- Backend (Logic Layer) – Processes API requests, managers OpenAI translations via LangChain, and returns JSON responses to the client.
- Database (Data Layer) – Maintains user accounts and translation logs within Supabase (PostgreSQL).

- Security and Deployment Layer – Cloudflare Tunnel encrypts all traffic and provides HTTPS routing with DDoS mitigation.

This overview was created using the N-tier layered architectural model (client-server) and documented with the C4 model.

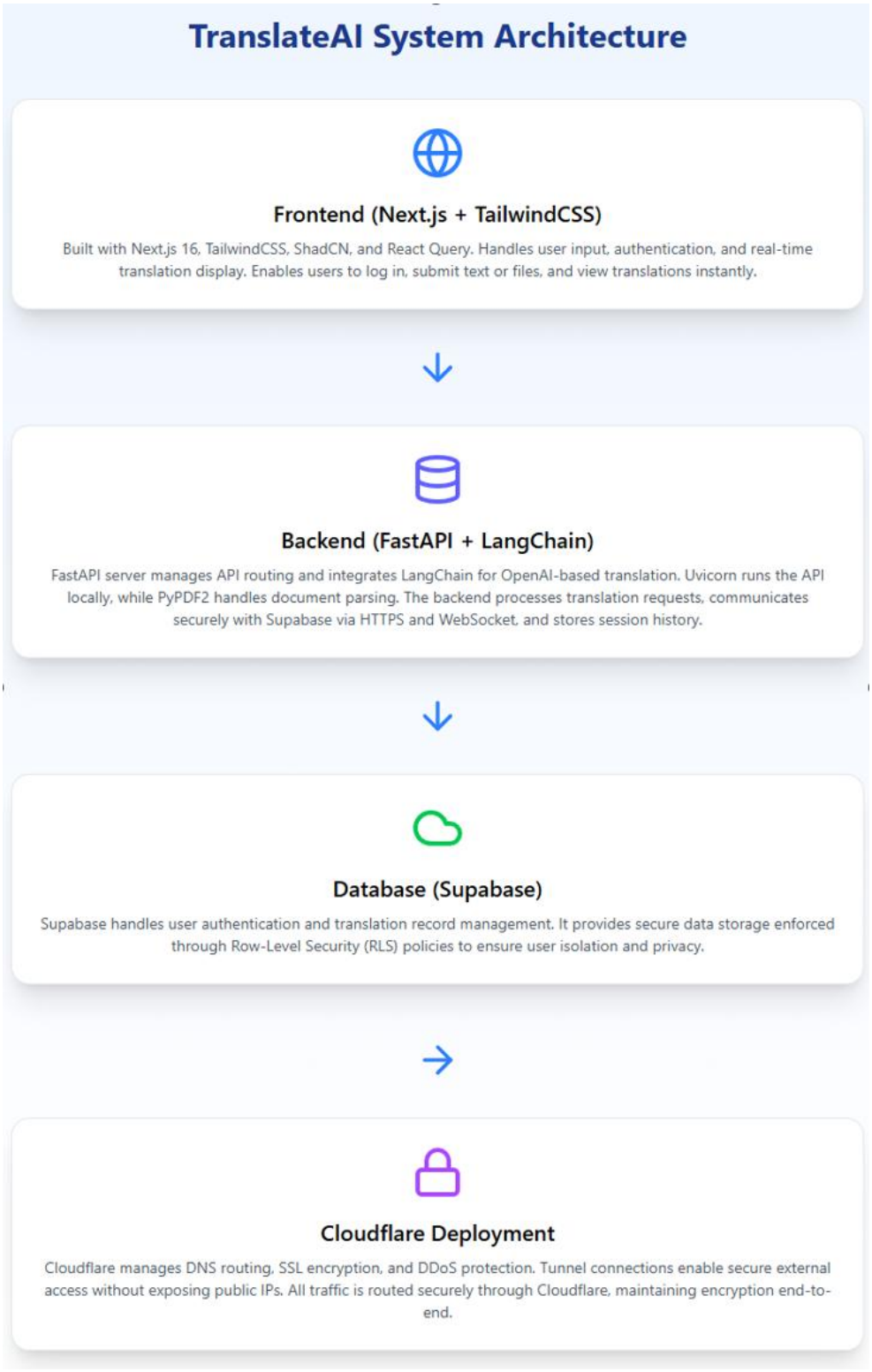


Figure 4. TranslateAI System Architecture depicting data flow from frontend to backend to Supabase via Cloudflare Tunnel.

1.5 Development Context

Local development was done to maintain strict control over environment variables and API keys. The OpenAI key was stored within a secure ".env" file, excluded from version control to prevent accidental leaks. Daily progress records (26 -29 October 2025) show the incremental implementation of authentication, translation modules, and Cloudflare integration. Backups of both codebases were maintained locally and via GitHub.

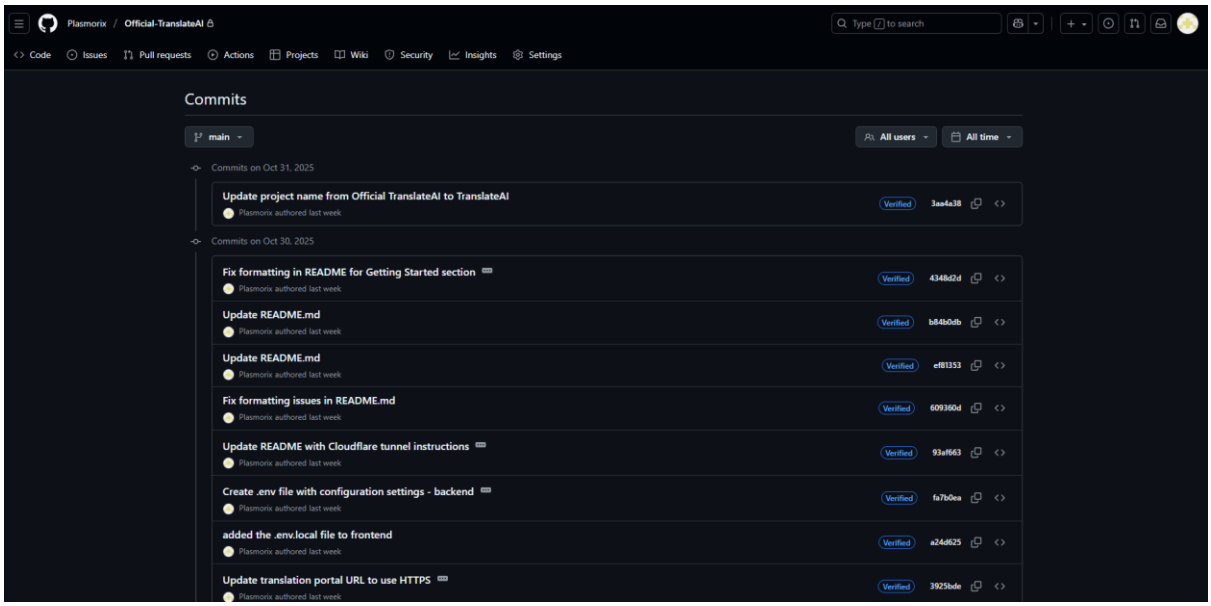


Figure 5. Outdated private GitHub repository showing commits.

2. Key Components of TranslateAI

2.1 Overview of the Architecture

System Design:

TranslateAI follows a modernised client-server architecture which is primarily designed for clarity, scalability, and secure integration. It is composed of a Next.js 16 frontend and a FastAPI (Python 3.12) backend. The client communicates with the backend through RESTful HTTPS requests, routed via Cloudflare Tunnels for encryption and DDoS protection. Supabase's PostgreSQL and Row-Level Security (RLS) handle the persistent storage and authentication.

When TranslateAI's source code is situated on a local client's PC, their PC can access TranslateAI via localhost. However, for all other users who have not downloaded the source code, the platform is deployed under a unified domain structure, as follows:

- translateai.nachiketp.me – Frontend web application
- translate-api.nachiketp.me – Backend API

Technologies Used:

Subsystem	Technology Stack	Purpose
Frontend	Next.js 16, TailwindCSS, ShadCN, React Query	Responsive UI and API communication
Backend	FastAPI, LangChain, PyPDF2, OpenAI API, WebSocket	Core translation logic and AI integration
Database	Supabase (PostgreSQL) and Row Level Security (RLS)	Authentication and data storage
Deployment	Cloudflare Tunnel, DNS, and SSL	Secure global access with no port exposure
Email Service	Brevo SMTP	Verification and password reset emails
Authentication	Supabase OAuth (Google & GitHub)	Secure social sign-in

2.2 Modules and Key Functions

TranslateAI is divided into two main subsystems:

- Frontend Subsystem (Next.js) – Responsible for user interaction, requests, and display of translated results.
- Backend Subsystem (FastAPI) – Responsible for AI processing, translation logic, and database interactions

2.3 Frontend Subsystem (Next.js 16, TailwindCSS, & ShadCN)

2.3.1 Purpose and Responsibilities

The frontend implements the presentation and interception layer for text, voice, image, and document translation. It:

- Captures user input (forms, uploads, mic)
- Authenticates via Supabase (Email/Password, Google, GitHub)
- Calls backend REST/WebSocket endpoints
- Manages request/response state and optimistic UI
- Renders results and history for the authenticated user.

2.3.2 File Location and Structure

https://github.com/Plasmorix/ITCapstone2_TranslateAI_NachiketPatel/tree/main/translator-frontend

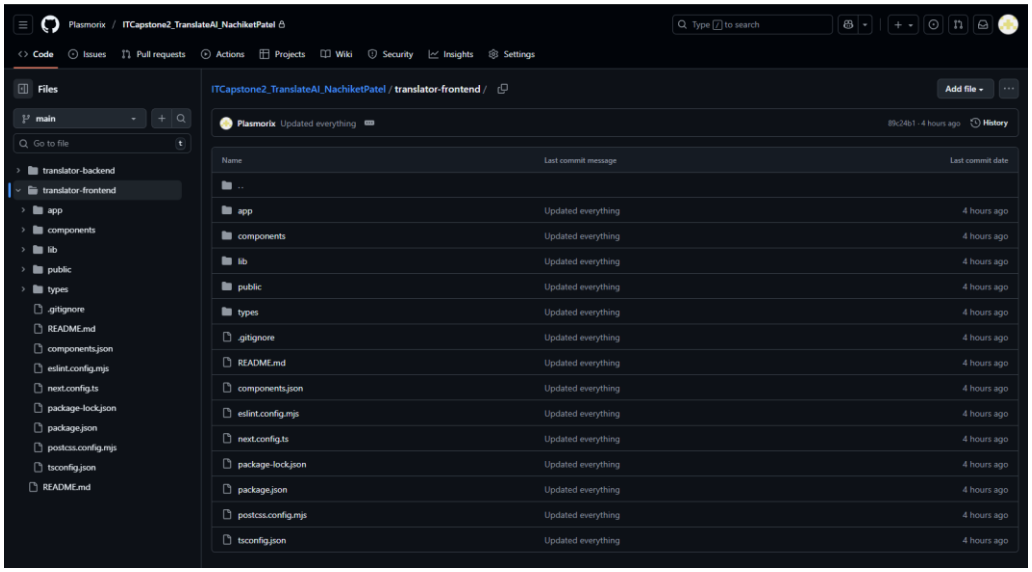


Figure 6. GitHub showing the actual folders/files of TranslateAI's frontend.

2.3.3 React Hooks and Server Actions (state and async data)

React hooks are used for efficient state management. An example of this could include keeping the UI reactive and predictable. On the other hand, server actions are used to handle asynchronous data, ensuring a smooth interaction between frontend components and backend APIs.

Code Snippets:

TextTranslation.txt

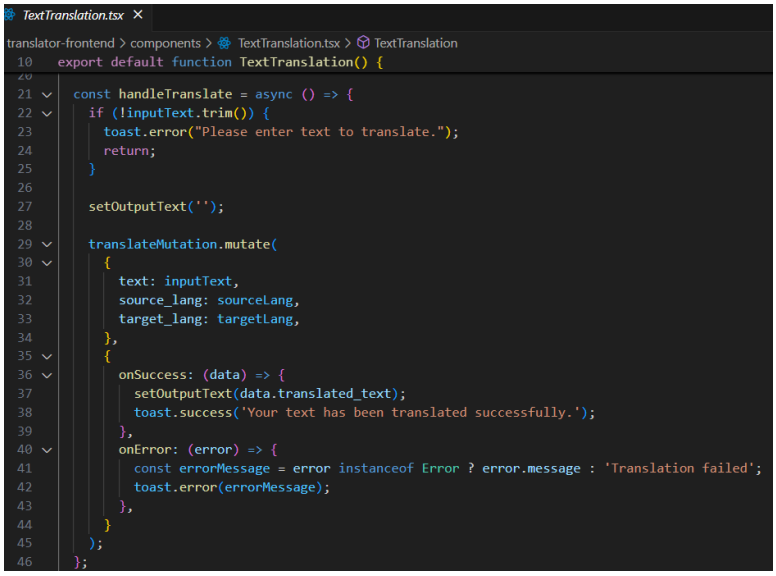


Figure 7. Text Translation's hook-driven mutate call converts input state into async request and processes results deterministically.

The “handleTranslate” function converts controlled input state into a single mutation call. UI feedback is optimistic and deterministic. It has a clear output, performs async fetch, then commits “translated_text” or surfaces an error toast. Hooks (useState, custom “useTextTranslation”) keep the flow predictable.

This code snippet is situated in /translator-frontend/components/TextTranslation.tsx.

Logic and Flow:

- Validate input → clear “outputText”
- “mutate({ text, source_lang, target_lang })”
- Success → set outputText; Error → toast

Parameters and Return Values:

- “handleTranslate()” → Promise<void>. Manages mutation and state.
- “translateMutation.mutate(payload)” → void; expects “{ translated_text: string }” in “onSuccess”

VoiceTranslation.tsx

VoiceTranslation.tsx

translator-frontend > components > VoiceTranslation.tsx > VoiceTranslation > connectWebSocket > useCallback() callback > onclose

```tsx
export default function VoiceTranslation() {

  const connectWebSocket = useCallback(async () => {
    console.log('🔌 Attempting to connect WebSocket...');

    try {
      const { data: { session } } = await supabase.auth.getSession();
      if (!session?.access_token) {
        toast.error('Authentication required for real-time translation');
        return;
      }

      const wsUrl = `${API_URL.replace('http', 'ws')}/v1/translate/audio/realtime?token=${session.access_token}`;
      console.log('🔗 Connecting to WebSocket:', wsUrl.replace(session.access_token, '[TOKEN]'));
      const websocket = new WebSocket(wsUrl);
      wsRef.current = websocket;

      websocket.onopen = () => {
        console.log('✅ WebSocket connected');
        setWsConnected(true);
        setStatus('Connected');
        setWs(websocket);
        toast.success('Connected to real-time translation service');

        const { sourceLang: currentSource, targetLang: currentTarget } = languageRef.current;
        websocket.send(JSON.stringify({
          type: 'config',
          source_lang: currentSource,
          target_lang: currentTarget
        }));
      };

```

Figure 8. VoiceTranslation establishes an auth'd WebSocket via Hooks, streams audio, and receives live transcription/translation events.

“connectWebSocket” (also known as “useCallback”) authenticates with Supabase, opens a WebSocket (WS) to the FastAPI service, and pushes a live configuration. Handlers (“onmessage”, “onerror”, “onclose”) drive a reactive state (“wsConnected”, “status”, transcript/translation), showcasing Hooks coordinating an async, bidirectional pipeline.

This code snippet situated in: /translator-frontend/components/VoiceTranslation.tsx.

Logic and Flow:

- Get session token → open WS → send { type: “config” }
- Stream audio (separate hook code) → handle server events → update UI
- Errors/close → clear refs, flip status.

Parameters and return values:

- “connectWebSocket()” → Promise<void> means side-effects on WS refs and connection stable
- “handleWebSocketMessage (event: MessageEvent)” → void, means it expects JSON with “type” keys like “input_transcription”, “translation”, etc

ImageTranslation.tsx

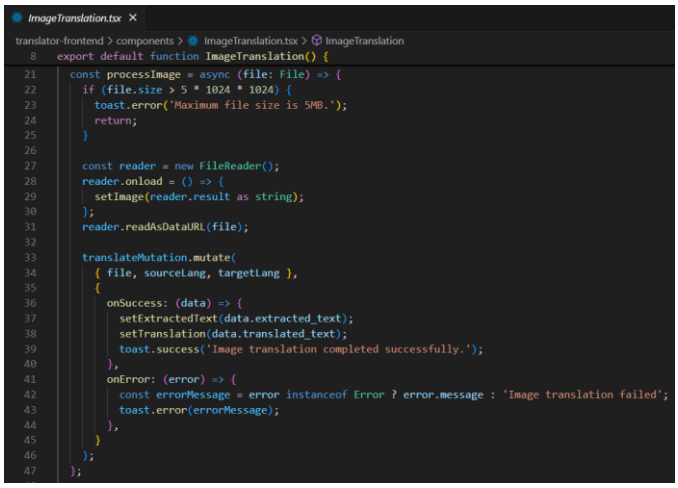


Figure 9. ImageTranslation combines local preview (FileReader) with an async mutation to return OCR text and translation.

"processImage" validates size, locally previews via "FileReader", then calls the mutation hook to perform OCR and translation. On success, it commits "extracted_text", demonstrating Hooks of UI consistency and a single async boundary,

This code snippet is situated in /translator-frontend/components/ImageTranslation.tsx.

Logic and Flow:

- Validate → Preview (data URL) → mutate
- Success → set "extractedText", "translation".
    - If an error occurs, a toast appears.

Parameters and return values:

- "processImage(file: File)" → Promise<void> portrays side-effects on "image", "extractedText", "translation".
- "TranslateMutation.mutate({ file, sourceLang, TargetLang })" → void; expects "{}"

DocumentTranslation.tsx

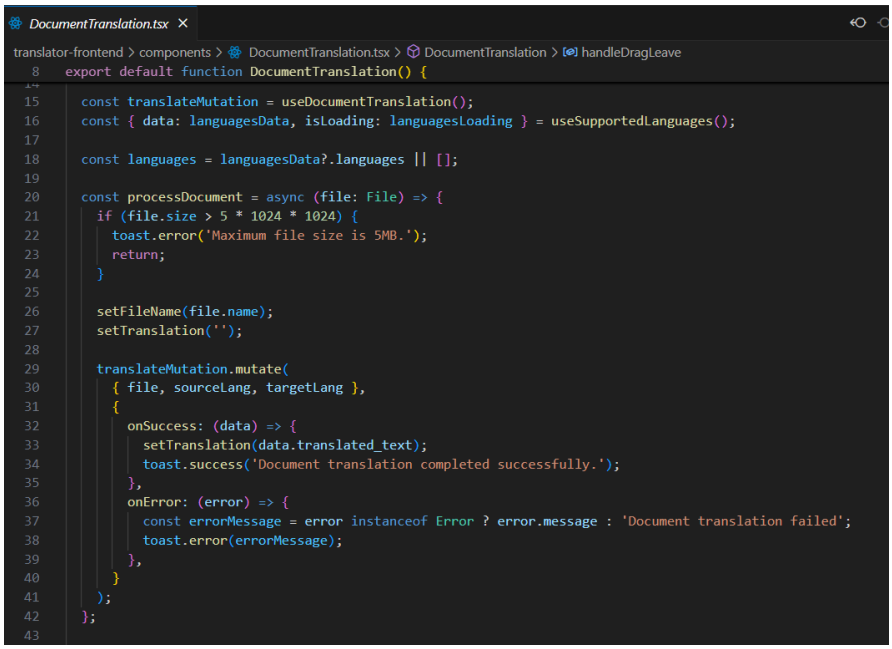


Figure 10. DocumentTranslation.tsx uses React state and a mutation hook to encapsulate aynchronous document translation and update the UI on success/error.

This component demonstrates Hooks for reactive UI ("useState") and a client fetcher (useDocumentTranslation) that wraps the asynchronous call. "processDocument" validates input, resets state, then triggers "mutate", updating UI via "onSuccess"/"onError" handlers. Supported languages are fetched via "useSupportedLanguages" keeping selections in sync with remote data.

This code snippet is in /translator-frontend/components/DocumentTranslation.tsx.

Logic and Flow:

- User drops or selects file → "processDocument(file)"
- Validate or reset → Call "translateMutation.mutate(payload)"
- Success→ set "translation" → toast; Error → Toast

Parameters and Return Values:

- "ProcessDocument(file:file)" → void. Side-effects: sets "filename", "translation"; invokes mutation.
- "translateMutation.mutate({ file, sourceLang, targetLang })" → void (react-query mutation). Result consumed via callbacks.
- onSuccess(data) expects "{ translated_text: string }"; updates "translation"

2.3.4 Reusable Components and Theming Consistency

TranslateAI utilised component-level reuse for UI and feedback. Buttons and Selects are wrapped once with brand tokens, motion, and accessibility via Tailwind, Radix, and “cva”, guaranteeing identical states and spaces across pages. The global “Toaster” joins theme, icons, and surface variables, so errors/success look and behave the same as in Text, Voice, Image, and Document flows. Input semantics such as password strength are centralised as small utilities and components that return deterministic outputs (scores/booleans) consumed by views. The main objective is for a clean separation of style, state, and semantics.

Examples of Reusable UI elements include:

Button.tsx (Primary theming surface)

anslator-frontend > components > ui > button.tsx > Button

```
const buttonVariants = cva(
  "inline-flex items-center justify-center gap-2 whitespace-nowrap rounded-md text-sm font-medium transition-all disabled:pointer-events-none disab
  {
    variants: {
      variant: {
        default: "bg-primary text-primary-foreground hover:bg-primary/90",
        destructive:
          "bg-destructive text-white hover:bg-destructive/90 focus-visible:ring-destructive/20 dark:focus-visible:ring-destructive/40 dark:bg-destr
        outline:
          "border bg-background shadow-xs hover:bg-accent hover:text-accent-foreground dark:bg-input/30 dark:border-input dark:hover:bg-input/50",
        secondary:
          "bg-secondary text-secondary-foreground hover:bg-secondary/80",
        ghost:
          "hover:bg-accent hover:text-accent-foreground dark:hover:bg-accent/50",
        link: "text-primary underline-offset-4 hover:underline",
      },
      size: {
        default: "h-9 px-4 py-2 has-[>svg]:px-3",
        sm: "h-8 rounded-md gap-1.5 px-3 has-[>svg]:px-2.5",
        lg: "h-10 rounded-md px-6 has-[>svg]:px-4",
        icon: "size-9",
        "icon-sm": "size-8",
        "icon-lg": "size-10",
      },
    },
    defaultVariants: {
      variant: "default",
      size: "default",
    },
  }
)
```

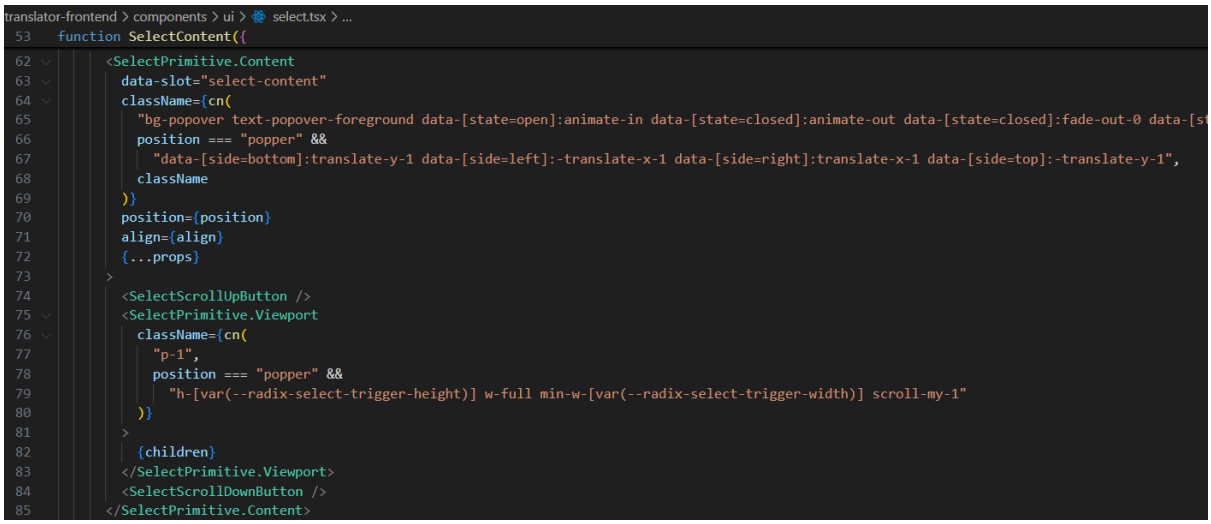
Figure 11. Central button variants via "cva" enforcing colour, spacing, and states.

This code snippet is in /translator-frontend/components/ui/button.tsx.

Logic and Flow:

Since “cva” composes all variants and sizes, every “<Button>” inherits brand tokens (TailwindCSS variables) with consistent hover/focus and disabled behaviour. Additionally, the consumers select “variant”/“size” without redefining styles.

Select.tsx (Language picker built on Radix)



```
translator-frontend > components > ui > select.tsx > ...
53  function SelectContent({
62      <SelectPrimitive.Content
63        data-slot="select-content"
64        className={cn(
65          "bg-popover text-popover-foreground data-[state=open]:animate-in data-[state=closed]:animate-out data-[state=closed]:fade-out-0 data-[s
66          position === "popper" &&
67            "data-[side=bottom]:translate-y-1 data-[side=left]:-translate-x-1 data-[side=right]:translate-x-1 data-[side=top]:-translate-y-1",
68          className
69        )}
70        position={position}
71        align={align}
72        {...props}
73      >
74        <SelectScrollUpButton />
75        <SelectPrimitive.Viewport
76          className={cn(
77            "p-1",
78            position === "popper" &&
79              "h-[var(--radix-select-trigger-height)] w-full min-w-[var(--radix-select-trigger-width)] scroll-my-1"
80          )}
81        >
82          {children}
83        </SelectPrimitive.Viewport>
84        <SelectScrollDownButton />
85      </SelectPrimitive.Content>
```

Figure 12. The themed "Select" wraps Radix with shared surface, border, and motion tokens.

This code snippet is in /translator-frontend/components/ui/select.tsx.

A single wrapper unifies popover surfaces, typography, and enter/exit animations. Moreover, language pickers across Text/Image/Voice/Document reuse this, ensuring identical UX and accessibility.

Sonner.tsx (Toast – Centralised Feedback)

```
sonner.tsx
translator-frontend > components > ui > sonner.tsx > ...
13  const Toaster = ({ ...props }: ToasterProps) => {
14    const { theme = "system" } = useTheme()
15
16    return (
17      <Sonner
18        theme={theme as ToasterProps["theme"]}
19        className="toaster group"
20        icons={{
21          success: <CircleCheckIcon className="size-4" />,
22          info: <InfoIcon className="size-4" />,
23          warning: <TriangleAlertIcon className="size-4" />,
24          error: <OctagonXIcon className="size-4" />,
25          loading: <Loader2Icon className="size-4 animate-spin" />,
26        }}
27        style={
28          {
29            "--normal-bg": "var(--popover)",
30            "--normal-text": "var(--popover-foreground)",
31            "--normal-border": "var(--border)",
32            "--border-radius": "var(--radius)",
33          } as React.CSSProperties
34        }
35        {...props}
36      />
37    )
38  }
```

Figure 13. Global "Toaster" respects light/dark theme and injects branded icons and tokens.

This code snippet is in /translator-frontend/components/ui/sonner.tsx.

Logic and Flow:

All success, error, and info toasts across features route through one themed entry point (“sonner” and “next-themes”) so messaging stays consistent without per-page styling.

PasswordStrengthMeter.tsx

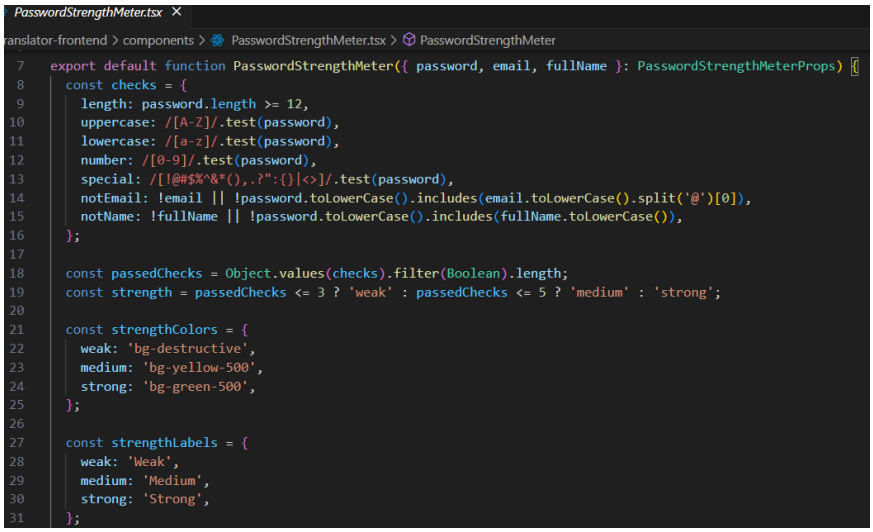


Figure 14. Password strength meter centralises the Regex policy and produces a stable, Weak, Medium, Strong score.

Note: Regex (Regular Expression) policy refers to a rule-based detection. (Security implementation)

This code snippet is in /translator-frontend/components/PasswordStrengthMeter.tsx.

Logic and Flow:

Forms consume a simple numeric score to process bars and labels, keeping security checks tight across authentication screens. Also, validation lives in a reusable meter.

2.3.5 API Abstraction and Error Handling (/lib/api.ts)

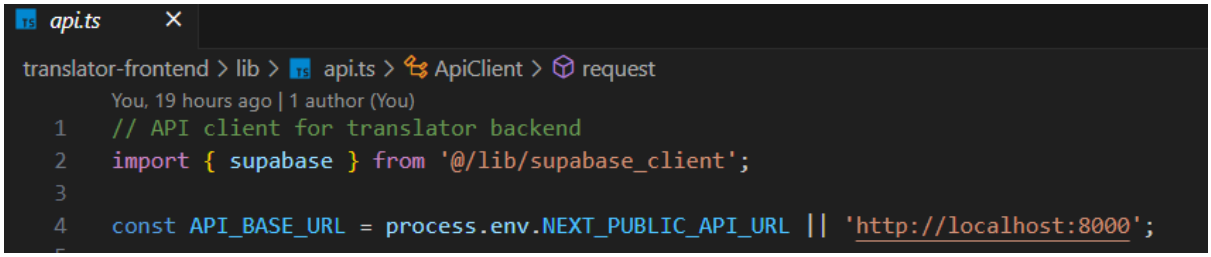


Figure 15 – Base URL comes from ".env" file

Figure 16. The "ApiClient" class centralises API communication by injecting Supabase JWT tokens into request headers and enforcing typed responses, ensuring secure and consistent backend interaction across all modules.

This code snippet is in /translator-frontend/lib/api.ts.

Logic and Flow:

The “api.ts” module defines strict TypeScript interfaces for translation responses, ensuring all API calls remain strongly typed. The custom “AuthError” class standardises authentication failures, while the “ApiClient” class centralises backend communication through an environment-based base URL. Each request retrieves a Supabase JWT via “getAuthHeaders()”, throwing an explicit “AuthError” if no valid session exists. When a token is present, it injects an “Authorization: Bearer <JWT>” header, ensuring all translation features communicate securely and consistently with the backend.

2.3.6 Security Implementation – Supabase JWT and OAuth

page.tsx

translator-frontend > app > auth > callback > page.tsx > ...

```
7   export default function AuthCallback() {
10    useEffect(() => {
11      const handleAuthCallback = async () => {
12        try {
13          const { data, error } = await supabase.auth.getSession();
14
15          if (error) {
16            console.error('Auth callback error:', error);
17            toast.error('Authentication failed. Please try again.');
18            router.push('/login');
19            return;
20          }
21
22          if (data.session) {
23            console.log('Auth callback successful:', data.session.user);
24            toast.success('Successfully signed in with Google!');
25            router.push('/dashboard');
26          } else {
```

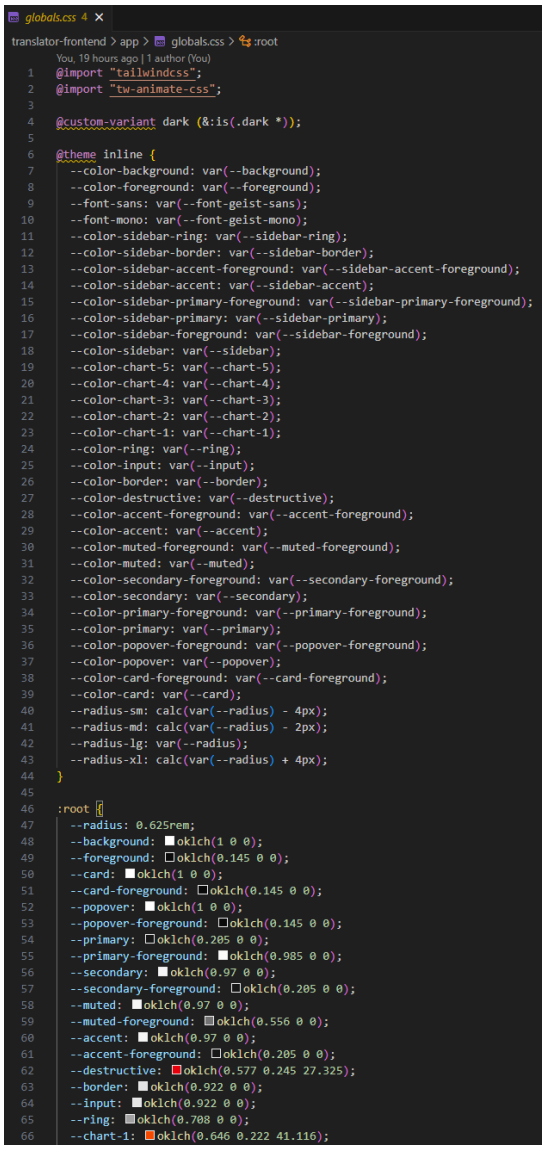
Figure 17. This figure shows authentication flow (JWT session and OAuth success/error feedback). Clean and visual.

This code snippet is in /translator-frontend/app/auth/callback/page.tsx

Logic and Flow:

“useEffect” runs “handleAuthCallback()”, which calls “Supabase.auth.getSession()” to retrieve OAuth session (JWT). If an error is returned, it logs the error, shows an error toast, and redirects to “/login”. If a valid “data.session” exists, it logs the user, shows a success toast, and redirects to “/dashboard”. If neither condition is met, it falls back to the login route. This ensures the app only moves forward once a verified Supabase JWT is present.

2.3.7 Styling and OKLCH Colour Tokens



```
globals.css 4
translator-frontend > app > globals.css > :root
You, 19 hours ago | 1 author (You)
@import "tailwindcss";
@import "tw-animate-css";

@custom-variant dark (&:is(.dark *));

@theme inline {
  --color-background: var(--background);
  --color-foreground: var(--foreground);
  --font-sans: var(--font-geist-sans);
  --font-mono: var(--font-geist-mono);
  --color-sidebar-ring: var(--sidebar-ring);
  --color-sidebar-border: var(--sidebar-border);
  --color-sidebar-accent-foreground: var(--sidebar-accent-foreground);
  --color-sidebar-accent: var(--sidebar-accent);
  --color-sidebar-primary-foreground: var(--sidebar-primary-foreground);
  --color-sidebar-primary: var(--sidebar-primary);
  --color-sidebar-foreground: var(--sidebar-foreground);
  --color-sidebar: var(--sidebar);
  --color-chart-5: var(--chart-5);
  --color-chart-4: var(--chart-4);
  --color-chart-3: var(--chart-3);
  --color-chart-2: var(--chart-2);
  --color-chart-1: var(--chart-1);
  --color-ring: var(--ring);
  --color-input: var(--input);
  --color-border: var(--border);
  --color-destructive: var(--destructive);
  --color-accent-foreground: var(--accent-foreground);
  --color-accent: var(--accent);
  --color-muted-foreground: var(--muted-foreground);
  --color-muted: var(--muted);
  --color-secondary-foreground: var(--secondary-foreground);
  --color-secondary: var(--secondary);
  --color-primary-foreground: var(--primary-foreground);
  --color-primary: var(--primary);
  --color-popover-foreground: var(--popover-foreground);
  --color-popover: var(--popover);
  --color-card-foreground: var(--card-foreground);
  --color-card: var(--card);
  --radius-sm: calc(var(--radius) - 4px);
  --radius-md: calc(var(--radius) - 2px);
  --radius-lg: var(--radius);
  --radius-xl: calc(var(--radius) + 4px);
}

:root {
  --radius: 0.625rem;
  --background: oklch(1 0 0);
  --foreground: oklch(0.145 0 0);
  --card: oklch(1 0 0);
  --card-foreground: oklch(0.145 0 0);
  --popover: oklch(1 0 0);
  --popover-foreground: oklch(0.145 0 0);
  --primary: oklch(0.205 0 0);
  --primary-foreground: oklch(0.985 0 0);
  --secondary: oklch(0.97 0 0);
  --secondary-foreground: oklch(0.205 0 0);
  --muted: oklch(0.97 0 0);
  --muted-foreground: oklch(0.556 0 0);
  --accent: oklch(0.97 0 0);
  --accent-foreground: oklch(0.205 0 0);
  --destructive: oklch(0.577 0.245 27.325);
  --border: oklch(0.922 0 0);
  --input: oklch(0.922 0 0);
  --ring: oklch(0.708 0 0);
  --chart-1: oklch(0.646 0.222 41.116);
```

Figure 18. The image above shows "globals.css" which defines TranslateAI's colour and radius tokens using OKLCH values, ensuring a consistent theme across the interface.

This code snippet is in /translator-frontend/app/globals.css

Logic and Flow:

The file links to TailwindCSS's theme utilities to custom OKLCH tokens, like "—primary", "--accent", and "--foreground", creating a completely unified design system. These tokens maintain visual balance, accessibility, and responsive colour behaviour, while the radius variables standardise the rounding of components for a consistent look throughout TranslateAI.

2.3.8 Frontend-Backend Interaction Flow

Figure 19. "translateText()" sends an authenticated HTTPS POST to the FastAPI endpoint via a shared "request" helper, with the Supabase JWT attached in headers.

This code snippet is in /translator-frontend/lib/api.ts.

Logic and Flow:

When the user submits a translation, the client assembles the payload and calls “translateText()”. This method fetches a Supabase session, injects the JWT into the “Authorization” header via “getAuthHeaders()” and posts to “/v1/translate/text” through the common request function. The backend (reached via Cloudflare Tunnel) returns JSON, which the frontend processes and logs to Supabase history.

2.4 Backend Subsystem (FastAPI, LangChain, and Supabase)

The backend is the logical core of TranslateAI. It authenticates users, validates requests, manages LLM translation via LangChain, persists history to Supabase, and exposes a clean HTTP API. The services follow a modular structure, favouring specific dependencies, and typed schemas.

2.4.1 File Location and Structure

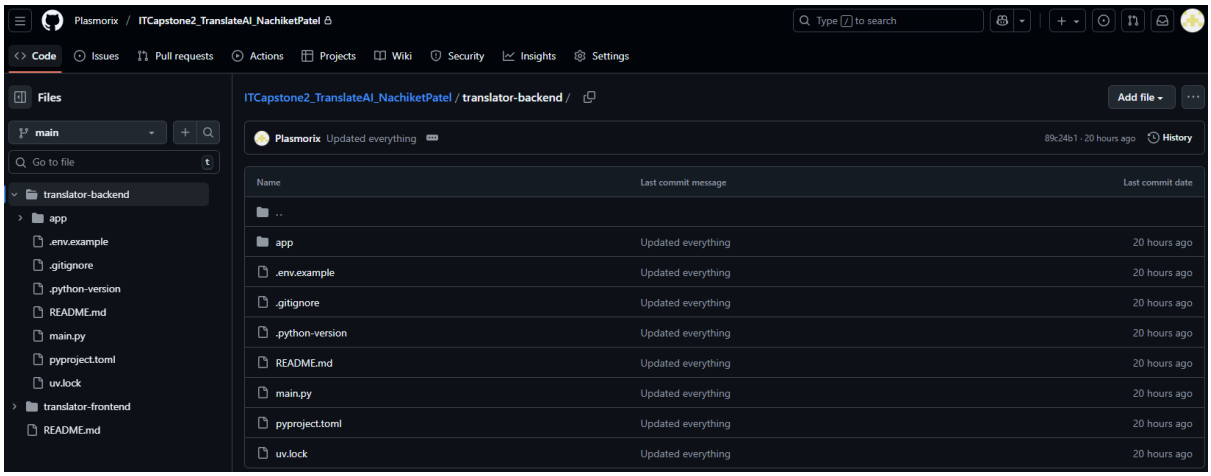


Figure 20. GitHub Repository showing Backend folders/file: https://github.com/Plasmorix/ITCapstone2_TranslateAI_NachiketPatel/tree/main/translator-backend

2.4.2 API Routing and Entry Point (app/main.py)

Figure 21. FastAPI entry point with middleware configuration (CORS), API routing, and health check endpoints for the Translator Backend.

This code snippet is in /translator-backend/app/main.py

Logic and Flow:

The application is created only after, CORS is configured to accept only known origins, and each domain router is mounted under a versioned prefix. Every request reaches the appropriate router while obtaining global middleware.

2.4.3 Authentication and Token Verification

All protected endpoints contain a dependency that enforces the presence of a Bearer token. In production, the token is verified against Supabase. Then, the dependency exposes user claims to route handlers.

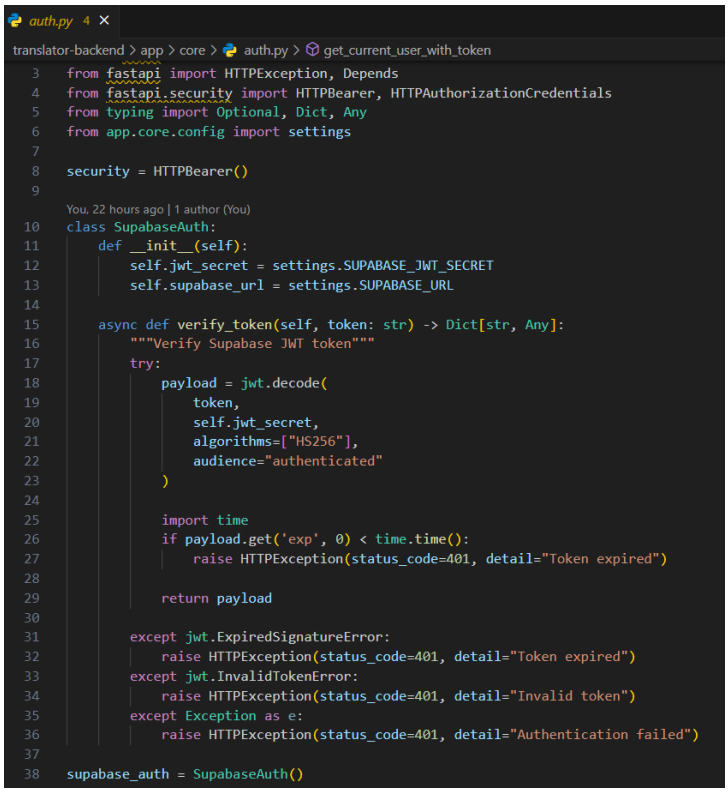


Figure 22. Request dependency enforcing Bearer token authentication.

This code snippet is in /translator-backend/app/dependencies/auth.py

Logic and Flow:

The dependency extracts the “Authorization” header, validates the scheme, and yields minimal identity to route handlers. Invalid or missing tokens immediately return HTTP 401, preventing unauthenticated execution.

2.4.4 Translation Workflow and Core Logic

Text translation demonstrates the core pattern: verify token → call LangChain service → time the call → persist history → return typed JSON.

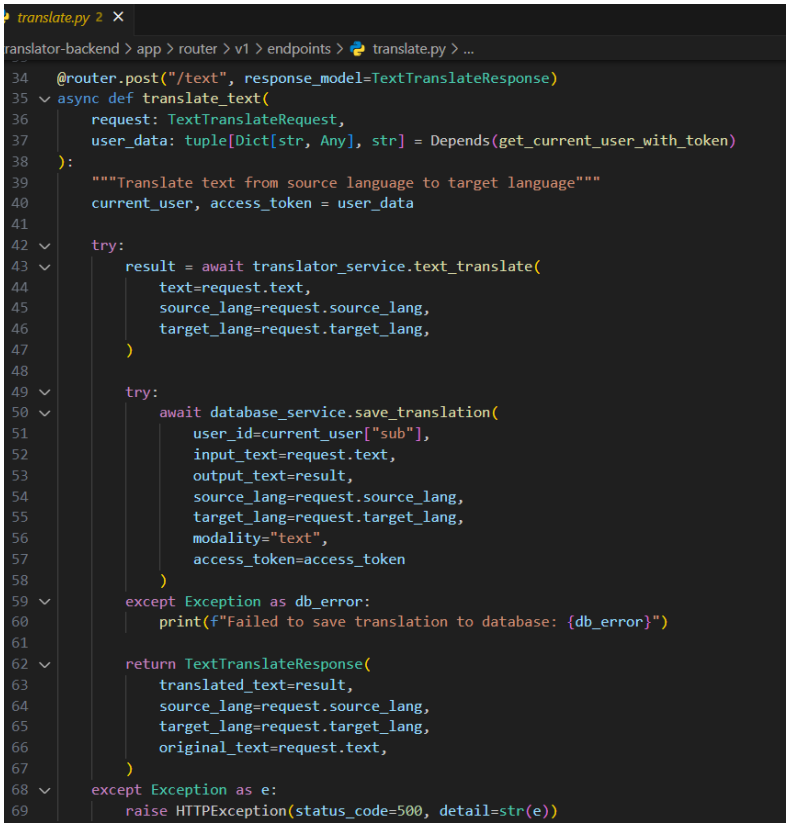


Figure 23. Authenticated text translation endpoint, managing LLM call and persistent history write.

This code snippet is in /translator-backend/app/router/v1/endpoints/translate.py

Logic and Flow:

The verified request hits the "/v1/translate/text", then the token is resolved through the "Depends(get_current_user_with_token)". "translator_service.text_translate(…)" runs the LangChain/OpenAI translation, later the result is saved via "database_service.save_translation(…) with user ID, source, target language, and modality. The typed response "TextTranslateResponse" is returned while any exception triggers an HTTP 500 error.

Parameters and Return Values:

"TranslateIn{text, source_lang, target_lang} → "TranslateOut{translate_text, latency_ms}.

2.4.5 LangChain and OpenAI Integration

LangChain standardises prompt composition and manages model execution, ensuring structured communication with the OpenAI API.

translator.py 2

translator-backend > app > services > translator.py > TranslatorService > document_translate

```python
class TranslatorService:
    """Service for handling translation logic"""

    def __init__(self):
        self.llm = ChatOpenAI(
            model="gpt-4o",
            api_key=settings.OPENAI_API_KEY,
            temperature=0.1,
        )

    async def text_translate(
        self, text: str, source_lang: str, target_lang: str
    ) -> str:
        """Translate given text using LangChain with OpenAI"""

        if not is_supported_language(target_lang):
            raise ValueError(f"Unsupported target language: {target_lang}")

        if source_lang != "auto" and not is_supported_language(source_lang):
            raise ValueError(f"Unsupported source language: {source_lang}")

        target_lang_name = get_language_name(target_lang) if target_lang != "auto" else target_lang
        source_lang_name = get_language_name(source_lang) if source_lang != "auto" else source_lang

        if source_lang == "auto":
            prompt = f"""Detect the language of the following text and translate it to {target_lang_name}.
Only return the translated text, nothing else.

Text to translate: {text}"""
        else:
            prompt = f"""Translate the following text from {source_lang_name} to {target_lang_name}.
Only return the translated text, nothing else.

Text to translate: {text}"""

        message = HumanMessage(content=prompt)
        response = await self.llm.ainvoke([message])

        return response.content.strip()
```

Figure 24. LangChain /OpenAI service builds a minimal prompt, calls "ChatOpenAI.ainvoke", and returns a trimmed string for the router.

This code snippet is in /translator-backend/app/services/translator.py

Logic and Flow:

The service opens a “ChatOpenAI” client with “gpt-4o” at “temperature=0.1” and validates source/target codes. Next, it crafts a short prompt (auto-detect if needed), sends a “HumanMessage” with “ainvoke”, then returns the model’s content as a clean string. Any invalid language codes raise “ValueError” before the call.

2.4.6 Database Communication via Supabase REST

The server writes history using service-role credentials which are never exposed to the client. Row Level Security (RLS) confines read access by user identity.

Figure 25. Will be explained in Figure 26.'s caption.

Figure 26. Supabase client (user-JWT scoped) inserts a translation record server-side with typed fields and UTC timestamps.

This code snippet is in /translator-backend/app/services/database.py

Logic and Flow:

The service builds a Supabase client using the user's JWT in the header, validates inputs inside the router, then "save_translation" inserts a row into the "translations" table with "id", "user_id", texts, language fields, modality, and "created_at". If Supabase returns data, that's the saved record. If not, it raises an exception. The same JWT-scoped client is also used for reading and deleting records, ensuring that RLS restricts each use to their own data.

2.4.7 Email Verification via Brevo SMTP

Supabase handles all verification and password-reset emails through its built-in Auth service using Brevo SMTP credentials. This removes the need for any custom mailing code in the backend.

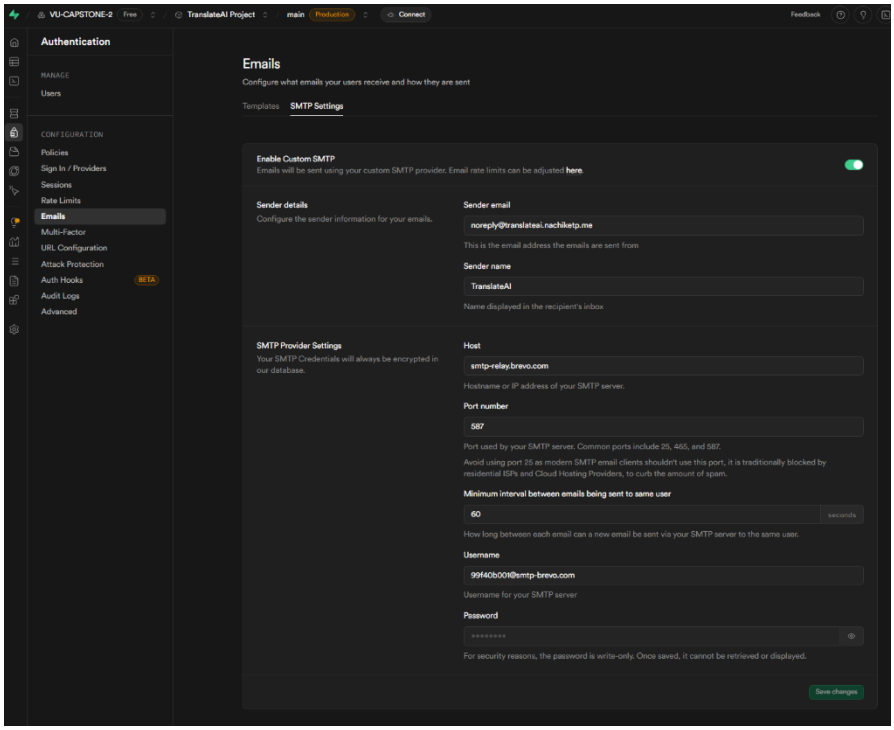


Figure 27. Supabase Auth SMTP settings configured with Brevo relay and custom verified sender domain.

Logic and Flow:

When a user signs up or resets a password, Supabase generates the token and sends the email via Brevo relay (smtp-relay.brevo.com). All templates and credentials are managed securely in Supabase, keeping the backend free of SMTP secrets.

2.4.8 Cloudflare Tunnel and Deployment

The backend binds to "localhost:8000" (Uvicorn) and is exposed securely via Cloudflare Tunnel (on VSC) on a public sub-domain (e.g., https://translate-api.nachiketp.me). Additionally, DNS is managed in Cloudflare and there is no router port-forwarding required.

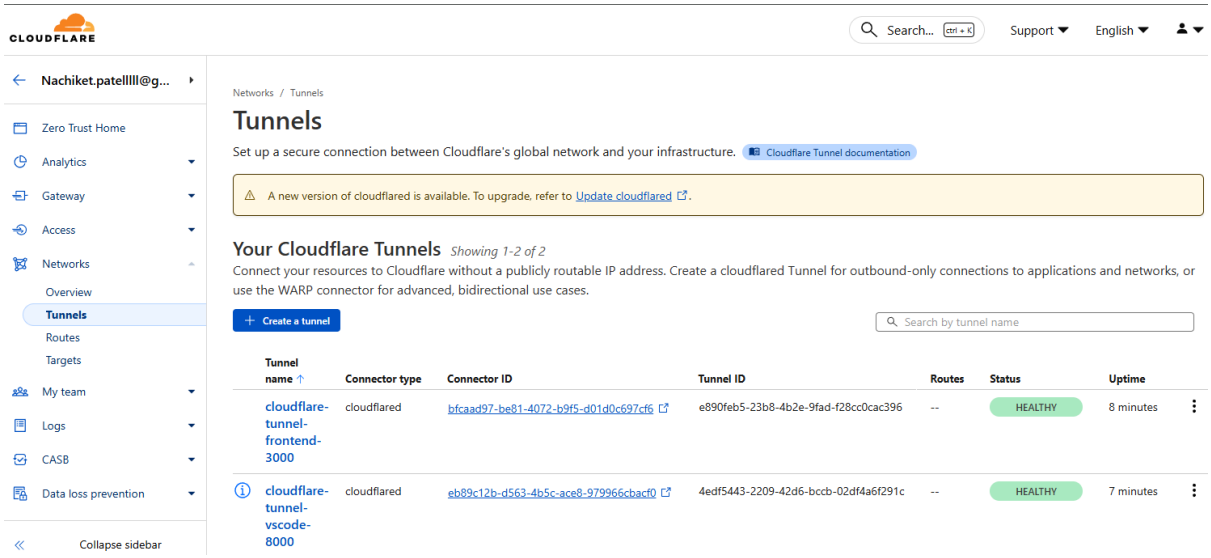


Figure 28. Cloudflare dashboard/CLI showing an active tunnel and the bound hostname.

Logic and Flow:

The nameservers are configured on Cloudflare. A sub-domain DNS record is created and then a Cloudflare tunnel is bound to Uvicorn. Next, WAF/SSL terminate at Cloudflare and HTTPS is proxied to the local service.

3. Implementation Strategy

3.1 Development Process

3.1.1 Scope

TranslateAI was implemented over a compact four-day sprints (26 – 29 October) using an iterative, time-capped workflow. Each day delivered a fully functional increment spanning the frontend (Next.js 16), backend (FastAPI), Supabase (Authentication and Database), and Cloudflare (DNS, Tunnel, and SSL). Additionally, the development followed a light Agile cycle, consisting of planning, implementing, verifying, and hardening. Validation on authentication, tunnelling, and RLS integrity was done on the final night.

3.1.2 Timeline

The following tasks were done on the following nights as per the project code:

26th October 2025: Foundation and First End-to-End (E2E) Test:

TranslateAI was initialised by setting up Next.js 16 frontend with TailwindCSS, ShadCN, and TweakCN for responsive styling. Routing, environment variables and initial translation UI components were configured along with Supabase authentication (email/password, Google, GitHub). On the backend, the FastAPI server was scaffolded with modular routed for text translation using LangChain and OpenAI GPT-4, integrated with Supabase for JWT verification. The first end-to-end test was conducted, and despite an OpenAI credit issue preventing translations, the frontend-to-backend flow was verified successfully.

27th October 2025 – Functional Translation and Expansion of the System:

Backend translation endpoints were completed for text, voice, image, document modes, integrating PyPDF2 for file extraction and WhisperAPI for audio transcription. WebSocket routed enabled real-time streaming. Supabase history logging and RLS verification were finalised. On the frontend, all translation types were connected to the API, ReactQuery optimised caching, and the application demonstrated full functionality during testing. OAuth login, password recovery, and protected routers were implemented. Additionally, the lecturer (Joseph) confirmed that all translation modes were working correctly, when the live demo was shown.

28th October 2025 – Cloud Integration and Stronger Security:

Cloudflare DNS and SSL were finalised, and Cloudflare Tunnel was configured to expose the FastAPI backend securely at “translate-api.nachiketp.me”. HTTPS communication between the frontend and backend was verified. Password-strength validation was added in Supabase,

and Auth0 was partially integrated but deprecated due to outdated compatibility. The frontend received its final visual refinements (including a “copy” button for translation outputs), mobile responsiveness was confirmed as working, and deployment under translateai.nachiketp.me became permanent for the Cloudflare Tunnel. Both tunnels and tokens were validated to ensure encrypted, token-based communication across all services.

29th October 2025 – Testing, Debugging, and Deployment Validation:

Cross-platform testing was conducted using Chrome, Microsoft Edge, and Safari (including an iPhone 14). Initial backend connectivity failed due to a missing tunnel, which was resolved by remapping Uvicorn port 8000 via the Cloudflare VSC extension. Following reconnection, end-to-end translation for text, voice, image, and document, functioned properly on laptops and mobile devices, confirming TranslateAI’s secure, scalable, and fully operational deployment.

3.2 Key Programming Concepts

3.2.1 Layered Architecture with Façade Services

The frontend communicates only through a single façade (/lib/api.ts), while FastAPI routes the incoming requests to service modules (translator.py, database.py). Supabase serves as an externalised data and authentication layer protected by RLS. Reasoning: it separates the presentation, logic, and data layers, allowing the UI to remain unaffected by modifications to the model or storage.

3.2.2 Contract-First DTOs (Pydantic and TypeScript)

Pydantic models define backend request/response contracts, and TypeScript types mirror the models on the frontend for type-safe fetches.

```python
from pydantic import BaseModel, Field
from typing import Optional


class TextTranslateRequest(BaseModel):
    """Request model for translation"""

    text: str = Field(..., description="Text to translate", min_length=1)
    source_lang: str = Field(
        ...,
        description="Source language (e.g., 'english') or 'auto' for automatic detection",
    )
    target_lang: str = Field("english", description="Target language (e.g., 'spanish')")


class TextTranslateResponse(BaseModel):
    """Response model for translation"""

    translated_text: str = Field(..., description="Translated text")
    source_lang: str = Field(..., description="Source language")
    target_lang: str = Field(..., description="Target language")
    original_text: str = Field(..., description="Original text")


class DocumentTranslateResponse(BaseModel):
    """Response model for document translation"""

    translated_text: str = Field(..., description="Translated document text")
    source_lang: str = Field(..., description="Source language")
    target_lang: str = Field(..., description="Target language")
    original_filename: str = Field(..., description="Original document filename")
    document_type: str = Field(..., description="Type of document processed")


class ImageTranslateResponse(BaseModel):
    """Response model for image translation"""

    extracted_text: str = Field(..., description="Text extracted from image")
    translated_text: str = Field(..., description="Translated text from image")
    source_lang: str = Field(..., description="Source language")
    target_lang: str = Field(..., description="Target language")
    original_filename: str = Field(..., description="Original image filename")
    image_type: str = Field(..., description="Type of image processed")
```

Figure 29. Pydantic models in translate.py defining request, and response schemas for text, document, and image translation. These classes enforce input validation and structured outputs in the FastAPI backend.

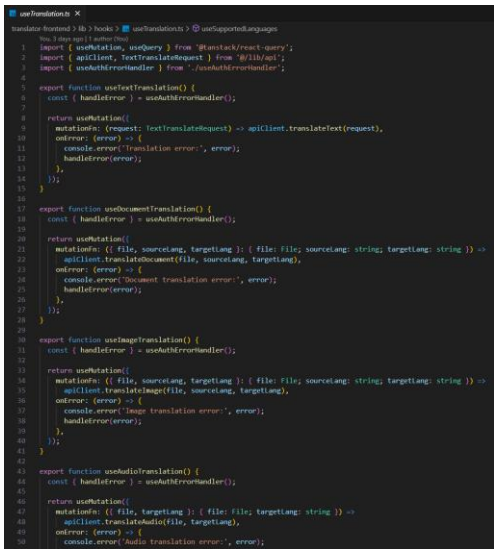


Figure 30. React Query mutation and query hooks in UseTranslation.ts, demonstrating strongly typed requests and responses to the backend. They ensure frontend data integrity and direct back to the backend Pydantic models.

3.2.3 Dependency Injection for Authentication (FastAPI Depends)

“verify_token” enforces JWT presence and yields user identity. Every protected route includes a “Depends(verify_token), ensuring uniform access control. Overall, it simplifies maintenance and reduces repetition.

3.2.4 Centralised HTTP Client (Frontend)

“ap.ts” defines a reusable “ApiClient” class that standardises all network requests using the Fetch API. It dynamically loads the base URL from the “.env” file, that attaches the Supabase JWT to every request and normalises error messages from the backend responses.

This design provides a single yet secure point of control for all translation calls (text, voice, image, and document) while maintaining minimal dependencies.

Figure 31. "api..ts" defines a reusable "ApiClient" that standardises authenticated network requests for all translation features.

3.2.5 React Query for Asynchronous State Management

React Query manages all backend communication versions through hooks that track request progress (“isPending”, “isError”, “isSuccess”) and automatically cache results. This eliminates manual loading logic and race conditions during continuous translations. It ensures the UI stays responsive and error feedback appears instantly across all translation modes.

3.2.6 OKLCH Token System for Theming (Globals.css)

The file “globals.css” defines uniform OKLCH colour tokens that ensure contrast and consistent styling in the light mode. These variables feed directly into the utilities of Tailwind, keeping design consistent without manual overrides. Radius tokens standardise the spacing across components.

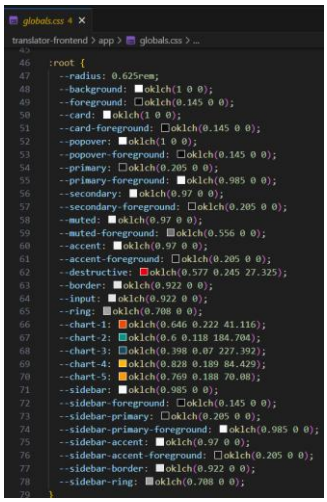


Figure 32. OKLCH colour tokens in "globals.css" providing consistent visual styling.

3.3 Integration of Functions

3.3.1 OpenAI Integration via LangChain (translator.py)

The file “translator.py” located in “translator-backend/app/schemas/translator.py, manages all OpenAI communication. It uses a reusable “TranslatorService” class that handles prompt generation, model activation, and post-processing for multiple input types. The services uses OpenAI (gpt-4o) for high accuracy, low latency, and dynamically builds prompts that detects language, preserves document structure, and returns clean outputs.

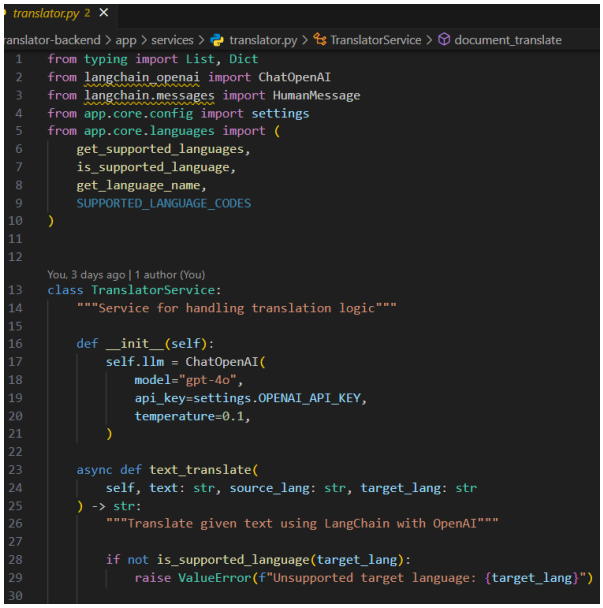


Figure 33. "TranslatorService" class in translator.py initialising the "ChatOpenAI" model and handling core translation logic.

3.3.2 Database Persistence via Supabase (database.py)

The “database.py” file handles write operations using Supabase’s REST API and a service-role key. Translation metadata such as user ID, mode, and latency are written asynchronously with secure headers. Row-Level Security (RLS) ensures each user can access only their own translation logs.

Figure 34. Supabase REST client in database.py securely writing authenticated translation logs using service-role credentials.

3.3.3 Secure Communication via Cloudflare Tunnel

Cloudflare Tunnel provides a secure HTTPS proxy between the local backend (running on port 8000) and the public subdomain "https://translate-api.nachiketp.me". This eliminated the need for port forwarding, and ensure all traffic is encrypted and DDoS-protected by Cloudflare's Web Application Firewall (WAF) layer.

Additionally, by using FastAPI, TranslateAI can exposes a local FastAPI instance securely to the internet during testing and demonstrations. This method also prevent direct IP discovery or IP leaks, significantly improving security for API endpoints.

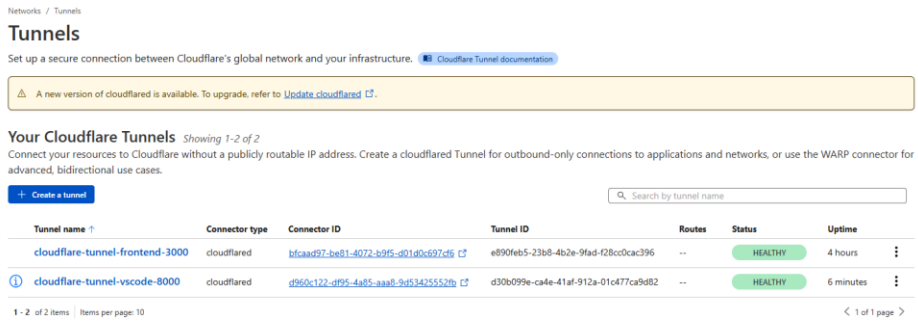


Figure 35. Active Cloudflare Tunnel connection securely routing backend traffic via HTTPS.

3.4 End-to-End Flow

When a translation request is initiated, several asynchronous processes occur in sequence. Each step operates independently but is linked through authentication and data flow integrity.

1. Frontend Validation: The UI verifies the inputs (text or file type) before sending data.
2. API Request: The "ApiClient" attaches the Supabase JWT and sends the request through Cloudflare Tunnel.
3. Backend Processing: FastAPI validates the JWT, runs the LangChain translation, and records latency.
4. Persistence: Translation data is securely written to Supabase with the server-only key.
5. UI rendering: React Query updates the interface and displays the translated output immediately.

3.5 Error Handling, Security, and Performance

3.5.1 Error Handling

All API methods include structured exception handling. Frontend toasts display readable messages from normalised backend responses. Each translation request creates an isolated environment, ensuring that any failure is contained and does not impact the overall interface.

3.5.2 Security

JWTs are verified for every request. Supabase RLS enforces per-user row access, and secrets remain in ".env" (never committed). TLS 1.3 encryption protects data end-to-end between the frontend and the backend.

```
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/03008580-637e-4eaf-b2e1-e1a5466ca119 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/03008580-637e-4eaf-b2e1-e1a5466ca119 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/03008580-637e-4eaf-b2e1-e1a5466ca119 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/03008580-637e-4eaf-b2e1-e1a5466ca119 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "OPTIONS /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 200 OK
INFO:     122.150.123.132:0 - "OPTIONS /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 200 OK
INFO:     122.150.123.132:0 - "OPTIONS /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 200 OK
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/03008580-637e-4eaf-b2e1-e1a5466ca119 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 200 OK
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 404 Not Found
INFO:     122.150.123.132:0 - "DELETE /v1/translate/history/4fd8ba2e-ec24-499c-9301-ceaed0993cb1 HTTP/1.1" 404 Not Found
```

Figure 36.HTTP responses showing successful and failed translation calls with descriptive error handling.

3.5.3 Performance

FastAPI's asynchronous I/O allows concurrent requests, while "gpt-4o" reduces model latency. Cloudflare's edge caching improves global response times. React Query caches successful translations to reduce repeat requests.

4. System Configuration and Setup

4.1 Configuration of the Setup

TranslateAI was developed and tested on Windows 11 Pro (64-bit) using Node.js v20+, Python 3.12+, and VS Code as the integrated environment. The system operated locally in two isolated workspaces, translator-frontend and translator-backend, which are connected through authenticated HTTPS and WebSocket requests.

4.1.1 Recommended Prerequisites:

- Node.js 20 or later
- Python 3.12 or later
- Git (2.42 +)
- Internet access to reach Supabase and OpenAI APIs

Cloudflare Tunnel configuration is optional and steps are covered in the User Manual for external deployment.

4.2 Installation Requirements and Configuration Files

1. Clone or download the official GitHub repository:
   git clone https://github.com/Plasmorix/ITCapstone2_TranslateAI_NachiketPatel
2. Create Environment files within the ".env" and ".env.local files"
3. Install dependencies ("npm i", "uv sync")

4.3 Deployment Process

1. Start Backend (FastAPI) using "uv run main.py"
2. Start Frontend (Next.js) using "npm run dev"
3. Access the local application via localhost:3000

5. Testing and Evaluation

5.1 Overview of the Testing Approach

Testing for TranslateAI focused on verifying that every translation mode – text, voice, image, and document performed accurately, consistently, and securely across multiple devices and browsers. Both functional testing (verifying expected outputs) and system integration testing (verifying communication between frontend, backend, and Supabase) were conducted.

All tests were performed locally under a Cloudflare-secured setup to replicate the final deployed environment, ensuring that authentication, API latency, and database logging behaved identically to the locally deployed version.

Testing took place from the 29th to the 31st of October 2025, following the final integration of the backend Cloudflare Tunnel.

5.2 Testing Strategies

TranslateAI's testing used a manual experimental approach supported by console logging and Supabase verification. The following methods were applied:

Type of Test	Purpose	Verification Method
Frontend Interaction Testing	To validate UI behaviour, form handling, and toast feedback across text, voice, image, and document translations	Performed manually through Chrome DevTools → Network Tab and Console Logs
Backend Response Testing	To confirm correct responses from FastAPI endpoints and verify authentication headers	Observed via VC Code integrated terminal logs (Uvicorn output).
Database Verification	To ensure user authentication and translation history are logged correctly in Supabase	Checked directly within the Supabase dashboard (Auth → Users and Tables → translation_history)
Cross-Device Validation	To confirm functionality and responsive layout on desktop and mobile browsers.	Tested manually on Windows 11 (Chrome & Edge) and iPhone 14 (Safari)
Performance and Error Checks	To measure average latency and confirm error toasts display for invalid input or network loss.	Measured using timestamps from backend logs and visible toast notifications.

5.2 Unit and Functional Testing

Frontend Functional Testing (Next.js 16)

Functional testing validated the full translation workflow for all input modes. Each scenario was tested within the deployed web application, ensuring consistent processing, accurate outputs, and correct toast and notifications.

Test Case	Input	Expected Output	Result
Text translation	"Hello, how are you?"	"你好, 你好吗？"	Working
Voice translation	Spoke "Hola, como estas?"	"Hello, how are you?"	Working
Image translation	Uploaded text image	Extracted OCR + translation	Working
Document translation	Uploaded document with text	Full translated text output	Working

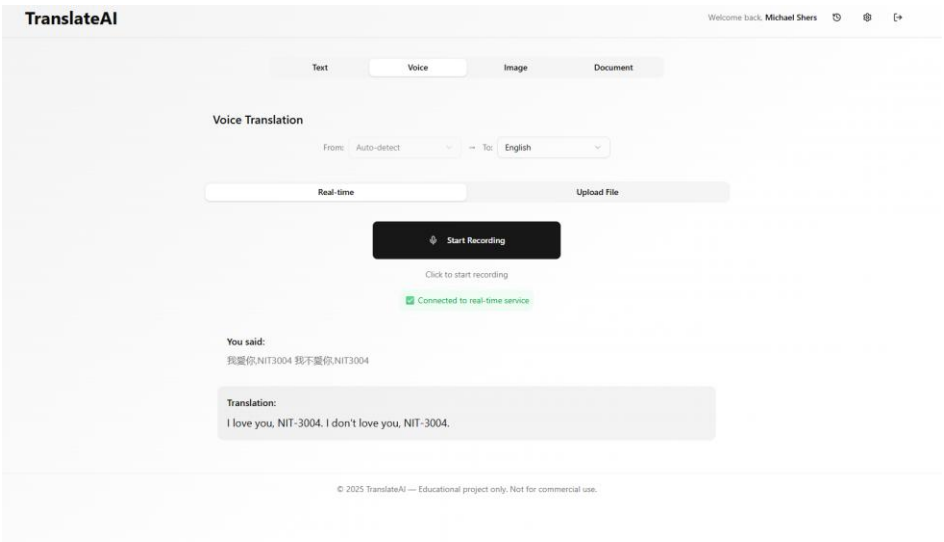


Figure 37. Real-time voice translation working successfully.

5.3 Integration and Performance Validation

After functional validation, integration testing was done to confirm successful communication between all layers – frontend, backend, Supabase, and Cloudflare.

Integration Results:

- Text translation: ~1.4s
- Voice translation (real-time): ~2.2s
- Document translation (PDF): ~3.1s

Image and Audio translation average API results varied significantly due to amount of text and file size.

5.4 Validation Criteria

TranslateAI was validated against the following criteria above derived from Capstone requirements:

Validation Category	Success Requirements	Result
Authentication	Users can register, login, and use Google/GitHub OAuth securely.	Working
Translation Accuracy	Outputs are theoretically correct across languages.	Working
File Upload Reliability	Image and document uploads do not exceed 5MB and produce results.	Working
Error Handling	The system displays toast notifications for all failed requests.	Working
Security	No unauthorised API access, and JWT tokens are validates at each endpoint	Working

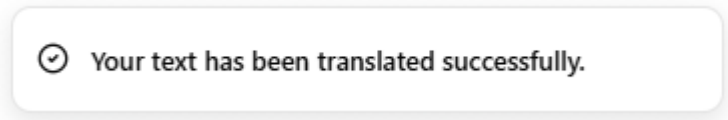


Figure 38. A toast notification successfully shown in the bottom-right corner of the page.

5.5 Cross-Platform Verification

Cross-platform verification showed consistent functionality across major operating systems and browsers:

- Windows 11 (Chrome and Edge): The rendering was smooth and no issues were present for CORS and latency.
- macOS and iOS (Safari): All translation modes functioned normally.

6. Challenges and Solutions

6.1 Overview of the challenges and solutions

Throughout TranslateAI's fast development cycle, several technical challenges appeared across authentication, backend configuration, and deployment management. Each issue was systematically investigated through documentation review, code reviewal, and iterative testing.

6.2 Challenge 1 – Auth0 Integration Failure

Problem:
Auth0 was initially selected as the authentication provider, however its SDK documentation relied on Next.js v13 syntax (Pages Router) syntax, whereas TranslateAI was built on Next.js v16 (App Router). Moreover, the callback URLs and API handler mismatches caused repeated "Invalid callback URL" and "Response not handled by middleware" errors to occur, creating conflicts with Supabase's existing authentication flow.

Solution:
Auth0 was completely removed and replaced by Supabase Auth, which provided smooth integration of the following:

- Email/password authentication
- OAuth logins via Google and GitHub
- Secure JWT sessions automatically refreshed on login

This transition reduced the overall complexity and enabled RLS to enforce per-user data isolation without extra middleware components.

6.3 Challenge 2 – OpenAI Credit and API Key Issues

Problem:
During early backend testing, all "/translate/…" endpoints returned HTTP401 Unauthorised errors. Further checkups within the code and overall found that the API key was valid, however the OpenAI account had no active credits within, hence preventing authentication.

Solution:
After recharging the account with a minimum balance, all requests authenticated successfully. Additionally, error handling was added in FastAPI to catch "OpenAIError", enabling there to be improvement in user feedback.

6.4 Challenge 3 – Cloudflare Tunnel Deployment Errors

Problem:
During testing in Safari on an iPhone 14, TranslateAI failed to load API responses from "translate-api.nachiketp.me", as the translate button was not functional. Moreover, the user was unable to select the output language. While laptop requests succeeded, the analysis revealed that the Cloudflare Tunnel had never been initiated, leaving the backend completely inaccessible.

Solution:
The backend tunnel was reconfigured using the Cloudflare VS Code extension, mapping port 8000 to the backend subdomain. The SSL handshake verification on the Cloudflare dashboard confirmed the connection. Moreover, this fix restored simultaneous operation of both frontend and backend over HTTPS without port forwarding or CORS violations.

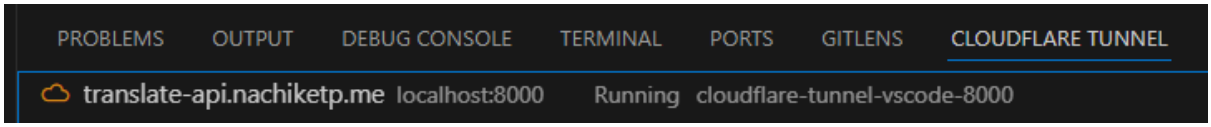


Figure 39. Active Cloudflare Tunnel linking the backend.

6.5 Challenge 4 – Supabase Sync and History Recording

Problem:
Initially, translation logs failed to appear in the "translation_history" table. Although, API responses were successful, no records were logged. This suggested that there was a possible mismatch for authentication.

Solution:
The issue was traced to the Supabase JWT key being exposed in the frontend ".env.local" file. The key was then moved exclusively to the backend ".env." file, while the frontend consisted of only the Supabase URL and anonymous authenticated user.

After resolving the issue, the backend writes securely with the service-role token, and RLS enforces read-only access per authenticated user.

6.6 Challenge 5 – Voice Translation Real-time Stability

Problem:
Voice translation occasionally dropped mid-session or failed to initialise due to Microphone Permission issues (within browser) and unstable WebSocket connections.

Solution:
A new "connectWebSocket()" function was implemented with error-recovery logic for reconnection and state management.
Status badges were added to the UI to provide immediate feedback:

- ✅ Connected to real-time service
- ⚠️ Disconnected – Connecting to real-time service…

This enhancement prevented silent failures and ensured a stable user experience. Microphone permissions were verified under HTTPS, which is required for WebRTC security.

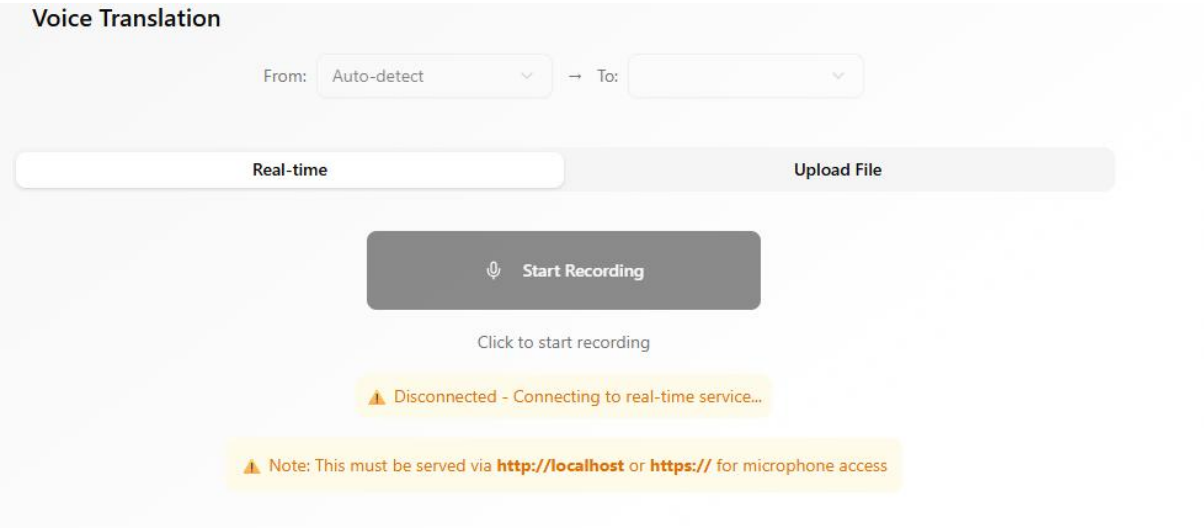


Figure 40. Disconnected Status

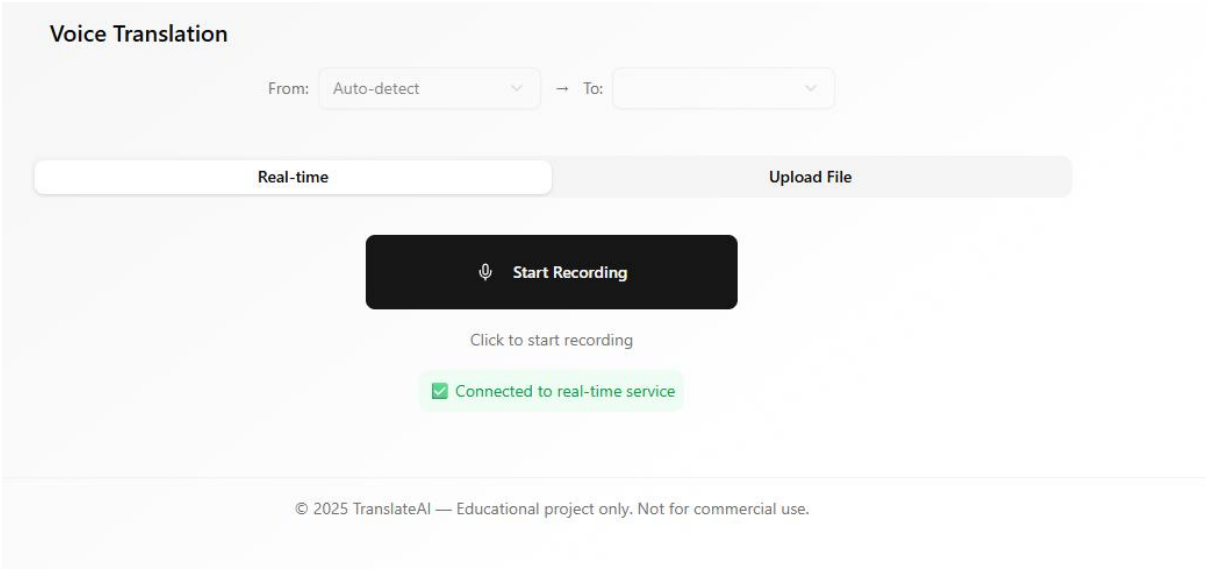


Figure 41. Connected status

7. Future Enhancements

7.1 Overview

TranslateAI has achieved its goal of providing secure multilingual translation for text, voice, image, and document inputs. As a result, future improvements will focus on speed, efficiency, accessibility, and scalability while preserving the original modular design.

7.2 Planned Improvements

1. **Model Upgrades and Funding:**
   Currently, TranslateAI operates on OpenAI's GPT-5o model using limited personal credit funding. Securing institutional or sponsor-based funding will enable sustained use of newer, more powerful models as they are released. This ensures that TranslateAI will provide more faster, smoother responses and have access to advanced capabilities such as better context reasoning.
2. **Streaming Translation:**
   Voice Translation currently processes buffered audio chunks. The next step is to introduce token-level streaming using OpenAI's streaming API so words appear live as users speak. This will delivery smoother, real-time conversations through incremental WebSocket messages.
3. **Accessibility and Offline Support:**
   Future versions could aim to add ARIA alerts for screen reader users. Enabling previously translated text and documents to remain available without an active internet connection could be significantly beneficial for service workers too.
4. **Improved Language Detection:**
   The planned integration of "langdetect" or OpenAI embeddings will improve automatic source-language identification. This pre-processing step will ensure higher translation accuracy and provide insights into user language trends.
5. **Layout-Preserving:**
   Upcoming updates can adopt "PyMuPDF" to reconstruct translated documents with their original structure, maintaining layout, formatting, and embedded imagery.

8. Conclusion

TranslateAI successfully demonstrates the integration of the latest web application technologies and artificial intelligence into a secure, multilingual translation platform. The system's design is based on a modular Next.js 16 frontend, a FastAPI backend, and Supabase for data authentication. It highlights how web architectures can deliver intelligent, yet real-time translation services. Using Cloudflare Tunnels, the application achieves both secure HTTPS routing and global accessibility, without public port exposure or complex hosting infrastructure.

From a technical perspective, the project showcases practical understanding of asynchronous data flows, RESTful, APIs, JWT-based authenticated tokens, and LLM-driven natural language processing using OpenAI and LangChain. Each subsystem was developed with an emphasis on maintainability, modularity, and performance consistency. The frontend's reliance on reusable UI components, custom hooks, and type-safe API abstraction reflects a disciplined approach to software design, while the backend's structure routing, schema validation, and logging mechanisms ensure robust service delivery.

The translation workflows involve text, voice, image, and document which represent a complete, functional environment where user input, authentication, processing, and output are compacted into a smooth user interface. The successful end-to-end communication between the frontend, backend, and database confirms the effectiveness of the entire implemented design choices and integration logic.

Despite the complexity of working a solo developer, this project met all the major milestones within an intense development timeline. The resulting application not only fulfils the original Capstone objectives but extends them by incorporating additional real-time and multimodal translation capabilities. Challenges faced in the authentication, latency, and tunnel configuration phases were overcome and resolved through systematic testing and adaptive development practices. Overall, it strengthened the technical depth of the implementation.

As for now, TranslateAI stands as a strong demonstration of applied cybersecurity, artificial intelligence, and full-stack IT engineering principles. With future developments planned to improve the model, caching, and accessibility, the platform is well positioned for continued innovation and scalability.

9. Acknowledgement

I would like to extend my gratitude to Dr Gongqi (Joseph) Lin for his continued academic guidance, and to Mr Csaba Veres, Unit Convenor for NIT3004 – IT Capstone 2, for his supervision and feedback throughout the development of TranslateAI. Their insights on the system design, integration, and documentation helped ensure that this project met both technical and professional standards.

Special thanks are also extended to Victoria University for providing the learning environment and resources for this Capstone project (disregarding the funding).

Lastly, I would like to acknowledge OpenAI for access its large-language-model APIs, which enabled the creation of a responsive, context-aware translation system, and Cloudflare for providing secure tunnelling and deployment services that ensured data privacy and reliability. Their platforms significantly contributed to TranslateAI's technical success.

10. References

GitHub, Inc. (2019). Build software better, together. GitHub. https://github.com/about

Microsoft. (2019). Visual Studio. Visual Studio. https://visualstudio.microsoft.com/

Microsoft 365. (n.d.). https://www.office.com/

Supabase. (n.d.). Supabase — The open source Firebase alternative. https://supabase.com/

OpenAI. (2024). OpenAI API platform. https://platform.openai.com/

LangChain. (n.d.). LangChain documentation. https://python.langchain.com/

FastAPI. (n.d.). FastAPI documentation. https://fastapi.tiangolo.com/

Uvicorn. (n.d.). Uvicorn — ASGI web server. https://www.uvicorn.org/

Render. (n.d.). Cloud hosting for developers. https://render.com/

Vercel. (n.d.). Vercel — Develop. Preview. Ship. https://vercel.com/

OpenAI. (2025). ChatGPT. https://chat.openai.com/