

YASSINE CHETTOUCH
VERTO · ENGINEERING

ENGINEERING · JANUARY 2026
TECHNICAL HANDOVER

Verto Engineering Handbook

ENG

PART I
Engine

PART II
Product

PART III
Runtime

APPENDIX
Services

YASCHET.COM
ENGINEERING HANDBOOK
001 OF 111

Contents.

Chapters	Page
Architecture overview	003
The engine	
Identity & security	008
Financial ledger	022
Intelligence pipeline	040
The experience	
User workspace	066
Governance & operations	087
Design engine & brand	093
Infrastructure	
Infrastructure & orchestration	102
Technical dependencies	110
Job runbook	062

Figures	Page
01 High-level topology	004
02 Authentication state machine	010
03 Middleware sequence	015
04 Ledger entity relationships	024
05 Financial data flow	032
06 Vision architecture	042
07 Pipeline lifecycle	044
08 Contextual batching	048
09 Editor composition	069
10 Regeneration loop	074
11 Realtime synchronization	081
12 Admin action sequence	089
Architecture decisions	063

ARCHITECTURE

Architecture overview

MAP

A map of the seven systems and their implementation boundaries.

IN THIS SECTION

01 System topology P. 004

02 Seven systems P. 005

03 Technology matrix P. 006

04 Codebase map P. 006

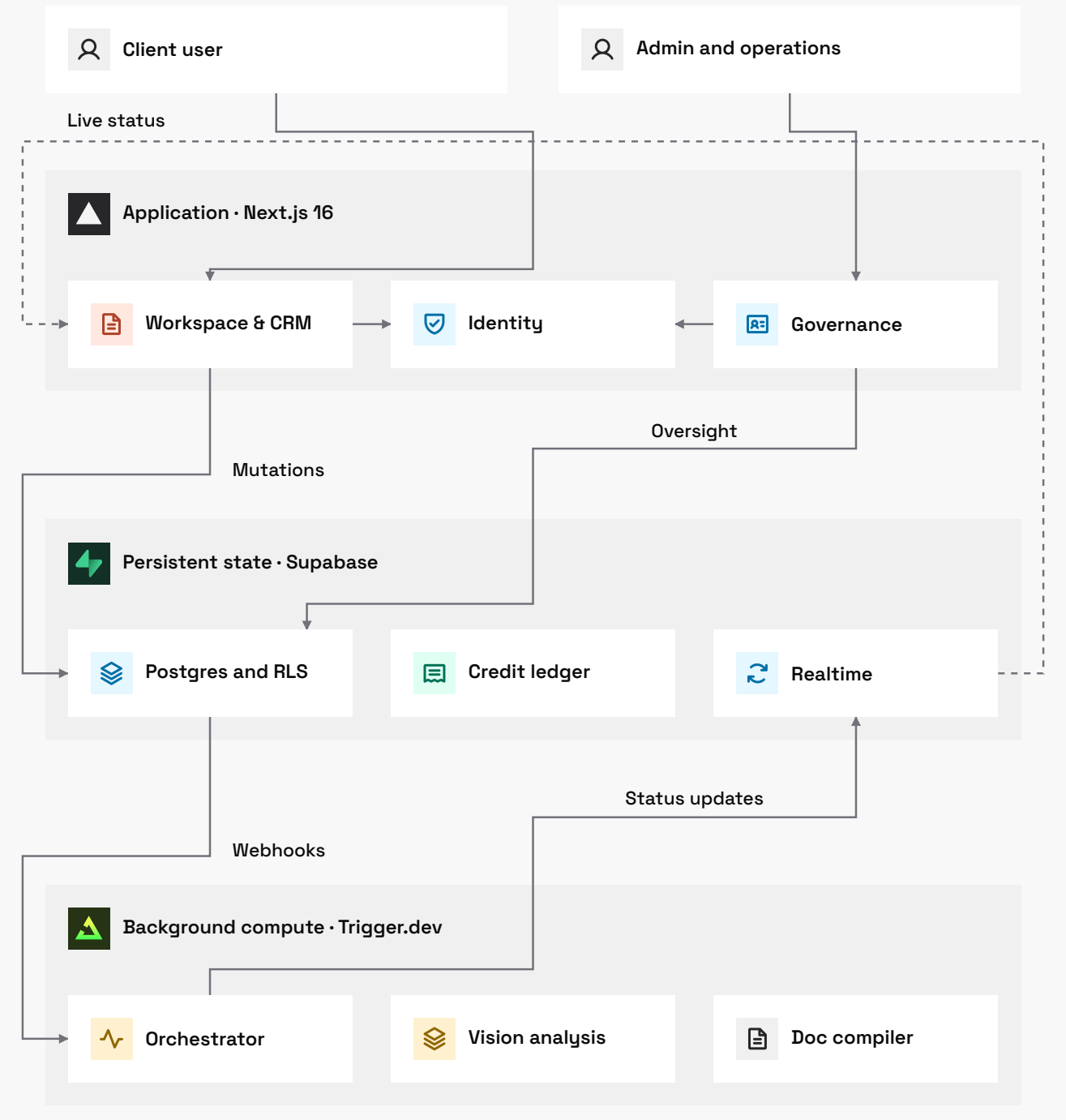
05 Developer quickstart P. 007

Verto translates documents from page images, preserving their structure in an editable document tree. Multimodal models interpret text, hierarchy, tables, and placement; the compiler reconstructs that tree in Google Docs for export.

This January 2026 technical handover explains seven systems, their implementation boundaries, and the procedures used to operate them.

1. System topology

01 High-level topology



2. Seven systems

Use the system map to find the owner of a concern. Each chapter then moves from responsibility to implementation, failure handling, and operational reference.

01 **Identity & security**

Authentication, MFA, sessions, and tenant data boundaries.

`src/actions/auth` · `supabase/migrations`

02 **Financial ledger**

Credit purchases, deductions, refunds, and balance history.

`src/actions/billing` · `src/utils/supabase/credits.ts`

03 **Intelligence pipeline**

Page images, translation, quality checks, and export.

`src/utils/trigger` · `src/utils/ai` · `src/utils/gcp`

04 **User workspace**

Document editing, source review, client records, and live state.

`src/components/translations/TranslationEditor` · `src/actions/services`

05 **Governance & operations**

Privileged actions, user recovery, credit adjustments, and reporting.

`src/app/(protected)/admin` · `src/actions/admin`

06 **Design engine & brand**

Product components, semantic tokens, notifications, and assets.

`src/components/ui` · `src/app/globals.css`

07 **Infrastructure & orchestration**

Hosting, persistent state, workers, networking, and deployment.

`src/env.ts` · `trigger.config.ts`

3. Technology matrix

LAYER	CORE TECHNOLOGY	SUPPORTING TOOLS
Frontend	**React**	shadcn/ui · Motion · Lucide · Tailwind CSS v4
State	**Server Actions**	TanStack Query · Zustand
Editor	**Tiptap**	ProseMirror · Headless document editing
Backend	**Trigger.dev v4**	Supabase JavaScript client
AI models	**GPT-5.2**	OpenAI SDK · Undici HTTP transport
Documents	**Google Docs**	Google APIs · Docs v1 · Drive v3

4. Codebase map

The codebase groups implementation by domain, with shared infrastructure beneath it.

SOURCE EXCERPT 1 · DIRECTORY TREE LINES 1–18

```
app/src/
├── actions/              # Server Actions (Mutations)
│   ├── auth/             # System 1: Identity (Sign-in, MFA)
│   ├── billing/          # System 2: Stripe & Credits
│   └── services/         # System 4: Customer CRM
├── app/                  # Next.js App Router (Routes & Pages)
│   ├── (protected)/      # Authenticated Dashboard
│   └── auth/             # Public Auth Flows
├── components/           # React Components
│   ├── branding/         # System 6: Branding Editor
│   └── translations/     # System 4: Editor & PDF Viewer
├── utils/                # Shared Logic
│   ├── ai/               # System 3: Intelligence Pipeline
│   ├── gcp/              # Google Docs Compiler
│   ├── supabase/         # Middleware & Clients
│   └── trigger/          # Trigger.dev Tasks
└── config/               # Static Configuration
    └── ai.ts             # Model definitions
```

5. Developer quickstart

Prepare the local environment, then use the commands below for data, migrations, and component review.

PREREQUISITES

- Node.js 20+ & pnpm
- Supabase CLI (via Homebrew)
- Stripe CLI

BOOT SEQUENCE

SOURCE EXCERPT 2 · BASH LINES 1–11

```bash
# 1. Start Local Infrastructure
pnpm supabase:start

# 2. Run the Development Server (Next.js + Turbopack)
pnpm run dev

# 3. Start Trigger.dev (Background Worker)
pnpm trigger:dev

# 4. Proxy Payments (In a separate terminal)
stripe listen --forward-to localhost:3000/api/payment/webhook
```

ESSENTIAL COMMANDS

Seed local data

`pnpm db:seed:dev`

Populate the local database with development users and translations.

Apply migrations

`pnpm supabase:migrate:local`

Apply SQL migrations to the local database.

Open component docs

`pnpm storybook`

Launch the component library.

PART I · THE ENGINE

Identity & security

01

Authentication, MFA, sessions, and tenant isolation.

IN THIS SECTION

01 Identity boundaries P. 009

02 Authentication flow P. 010

03 Multi-factor authentication P. 013

04 Session middleware P. 015

05 Tenant isolation P. 017

06 Identity lifecycle P. 018

07 Security controls and excerpt limits P. 021

Modules	`src/actions/auth/*` · `supabase/migrations/*`

1. Identity boundaries

Authentication separates identity from permission to use a protected route or action. A valid session is only the starting point; tenant, account-status, and assurance checks determine access.

Supabase Auth provides the identity and session primitives. Verto's server-side controls require a second factor where the application's MFA policy demands it.

ACCESS CHECKS

Session identity

Supabase Auth supplies identity and session primitives. A session is the starting point for access checks.

MFA assurance

Sensitive routes and actions must enforce the required assurance level. Password sign-in alone can remain at AAL1.

Data ownership

RLS policies bind ordinary translation access to the owning profile. Administrative access needs a separate authorized path.

Sensitive changes

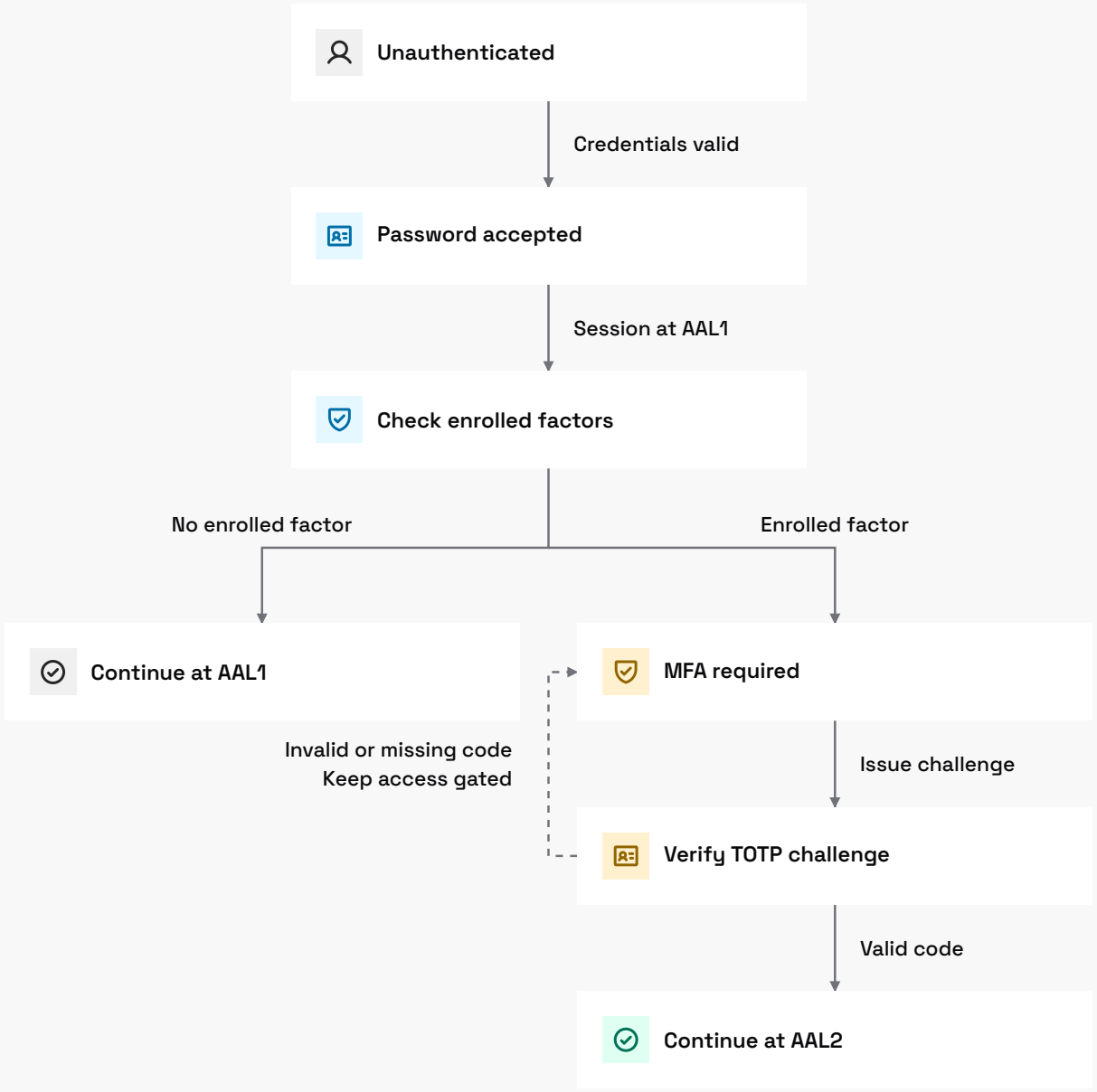
Password changes and account deletion use fresh password verification in the documented flows.

2. Authentication flow

Implementation `src/actions/auth/sign-in.ts`

The `signIn` action checks credentials and enrolled factors. The middleware example later in this chapter checks authentication assurance before allowing protected navigation.

02 **Authentication state machine**



Password sign-in and access assurance are separate: AAL1 does not satisfy an AAL2 gate.

2.1 Password sign-in and MFA gate

Password sign-in may establish an `aal1` session. When MFA is required, the application must keep sensitive routes and actions unavailable until the session reaches `aal2`.

01 Submit credentials

Sign in with the user's email and password.

02 Inspect assurance

Check the enrolled factors and the session's current and required authentication assurance levels.

03 Require verification

When the session requires `aal2`, route the user to the challenge and block protected actions until the second factor is verified.

SOURCE EXCERPT 3 · TYPESCRIPT LINES 1–26

```typescript
// src/actions/auth/sign-in.ts

const { data, error } = await supabase.auth.signInWithPassword({
    email: email,
    password: password,
});

if (error) {
    // ... handle error
}

const { session, user } = data;

// CRITICAL: The Halt State Interceptor
// If session is null but user exists, it means 2FA is required (AAL1 -> AAL2 escalation needed)
if (user?.id && user?.confirmed_at) {
    if (user?.factors && user?.factors.length > 0) {
        return {
            code: 'MFA_REQUIRED',
            description: 'Please enter the code sent to your email address.',
            message: 'Multi-Factor Authentication is required!',
            status: 401,
            success: true,
        };
    }
}
```

Session boundary

Password sign-in can create an `aal1` session. The source excerpt checks enrolled factors; the middleware excerpt checks current and required assurance. Do not infer MFA enforcement from a missing session alone.

2.2 Registration and profile creation

Implementation `src/actions/auth/sign-up.ts`

Registration creates the authentication identity and supplies metadata for the associated `public.profiles` record. The excerpt shows the sign-up call; the database trigger or transaction that completes profile creation is outside this listing.

SOURCE EXCERPT 4 · TYPESCRIPT LINES 1–18

```ts
// src/actions/auth/sign-up.ts
const registerUser = async (supabase: SupabaseClient, validFields: z.infer<typeof RegisterSchema>) => {
    const { data, error } = await supabase.auth.signUp({
        email: validFields.contactInformation.email,
        options: {
            data: {
                // These fields are automatically propagated to public.profiles via Trigger
                first_name: validFields.personalInformation.firstName,
                last_name: validFields.personalInformation.lastName,
                company_name: validFields.professionalInformation.companyName,
                role: 'user',
                is_profile_complete: true,
            },
        },
        password: validFields.accountSecurity.password,
    });
    // ... error handling
};
```

2.3 Authentication callback

Implementation `src/app/auth/callback/route.ts`

The server-side callback exchanges an OAuth or passwordless sign-in code for a session, then completes the application's callback handling.

Sign-in event

The callback handler records `sign_in` in `user_activity`, including callback-based social sign-ins. Check the insert result and retention policy when auditing activity coverage.

SOURCE EXCERPT 5 · TYPESCRIPT LINES 1–20

```ts
// src/app/auth/callback/route.ts
export async function GET(request: Request) {
    const { searchParams } = new URL(request.url);
    const code = searchParams.get('code');

    if (code) {
        // Atomic Exchange: Code -> Session
        const { error } = await supabase.auth.exchangeCodeForSession(code);

        if (!error) {
            // Compliance: Log the entry event
            await supabase.from('user_activity').insert({
                type: 'sign_in',
                details: { method: 'callback' },
            });
            return NextResponse.redirect(nextUrl);
        }
    }
    // ... error handling
}
```

3. Multi-factor authentication

Implementation `src/actions/auth/multi-factor-authentication.ts`

The documented second factor uses TOTP authenticator codes. Enrollment and verification raise session assurance from `aal1` to `aal2`; sensitive actions must check that assurance server-side.

3.1 Challenge and verification

Server actions submit MFA challenges and verify the response with the identity provider.

01 Fetch factors

Retrieve the `factorId` for the user's registered TOTP device.

02 Issue challenge

Call `supabase.auth.mfa.challenge({ factorId })`.

03 Verify code

Verify the six-digit code with `verifyChallenge`. A valid response upgrades the session from AAL1 (password) to AAL2 (MFA).

SOURCE EXCERPT 6 · TYPESCRIPT LINES 1–6

```ts
// src/actions/auth/multi-factor-authentication.ts
const { data, error } = await supabase.auth.mfa.verify({
    factorId,
    challengeId,
    code,
});
```

3.2 Sign-out and local cleanup

Implementation `src/actions/auth/sign-out.ts`

Sign-out clears the authentication session and the client query cache so data from the previous user does not remain visible in the workspace.

SOURCE EXCERPT 7 · TYPESCRIPT LINES 1–11

```ts
// src/actions/auth/sign-out.ts
export const signOut = async () => {
    // 1. Server-Side: Kill the Supabase Session
    const { error } = await supabase.auth.signOut();

    // 2. Client-Side: Purge Information cache
    // This ensures no previous queries (e.g., 'user-profile', 'translations') remain accessible.
    const queryClient = getQueryClient();
    queryClient.clear();
    await queryClient.resetQueries();
};
```

3.3 Factor enrollment

Enrollment requires a successful verification code before the factor is treated as ready for use.

01 Create secret

We create a QR code (SVG) on the server.

02 Confirm device

The factor is `unverified` until the user successfully submits their first code. Only then does the status flip to `verified`, enforcing the requirement for future logins.

4. Session middleware

Implementation `src/utils/supabase/middleware.ts`

`updateSession` refreshes authentication state and applies the middleware's route checks. Protected actions and database policies still need their own authorization boundaries.

4.1 Refresh before authorization

The middleware attempts to refresh authentication state before applying its route checks.

03 Middleware sequence



Session refresh and route authorization are separate checks.

4.2 Route and assurance checks

Implementation `src/utils/supabase/middleware.ts`

SOURCE EXCERPT 8 · TYPESCRIPT LINES 1–40

```ts
// src/utils/supabase/middleware.ts
export async function updateSession(request: NextRequest) {
    // 1. Initialize Client with Cookie Handling
    const supabase = createServerClient(..., {
        cookies: {
            get(name) { return request.cookies.get(name)?.value; },
            set(name, value, options) { supabaseResponse.cookies.set(name, value, options); },
            remove(name, options) { supabaseResponse.cookies.set(name, '', { ...options, maxAge: 0 }); },
        },
    });

    // 2. Resilience: Race Condition Handling for Refresh
    try {
        await Promise.race([
            supabase.auth.getSession(),
            new Promise((_, reject) => setTimeout(() => reject(new Error('Timeout')), 5000)),
        ]);
    } catch {
        // Fallback to degraded mode (check cache)
    }

    // 3. Status Enforcement (Suspension Check)
    if (user && userStatus === 'SUSPENDED') {
         await supabase.auth.signOut();
         return NextResponse.redirect(`${baseUrl}/auth/suspended`);
    }

    // 4. AAL Enforcement (MFA Check)
    const { data: mfaData } = await supabase.auth.mfa.getAuthenticatorAssuranceLevel();
    if (mfaData.nextLevel === 'aal2' && mfaData.nextLevel !== mfaData.currentLevel) {
        return NextResponse.redirect(`${baseUrl}/auth/mfa/challenge`);
    }

    // 5. Protected Route Guard
    if (isMatch(pathname, protectedRoutes) && !user) {
        return NextResponse.redirect(`${baseUrl}/auth/sign-in`);
    }

    return supabaseResponse;
}
```

5. Tenant isolation

Implementation `supabase/migrations/00000000000003_secure_rls.sql`

The documented access model enables RLS on application tables and permits access through explicit policies. Review the deployed policies and grants to establish the actual boundary.

5.1 Tenant access policy

A tenant must not read another tenant's documents. The translation policy binds ordinary user access to the owning profile.

Translation policy

SOURCE EXCERPT 9 · SQL LINES 1–10

```sql
-- migrations/03_secure_rls.sql
CREATE POLICY "Users can only view their own translations" ON "public"."translations"
    FOR ALL
    USING (
        (auth.uid() = user_id)
        -- Admin Policy overrides this at the table level with a separate policy
    )
    WITH CHECK (
        (auth.uid() = user_id)
    );
```

5.2 Administrative access

Administrative access requires a separate authorized policy or an elevated server client. The excerpt below shows the policy-based exception:

SOURCE EXCERPT 10 · SQL LINES 1–16

```sql
-- migrations/03_secure_rls.sql

-- 1. The Function (Robust Check: JWT + DB Fallback)
CREATE OR REPLACE FUNCTION public.is_admin() RETURNS BOOLEAN AS $$
DECLARE
  user_role public.app_role;
BEGIN
  -- Check JWT claim first (Fast path)
  IF (auth.jwt() ->> 'user_role')::public.app_role = 'ADMIN'::public.app_role THEN
    RETURN TRUE;
  END IF;

  -- Check Database (Reliable fallback)
  IF EXISTS (
    SELECT 1 FROM public.user_roles
    WHERE user_id = auth.uid()
```

SOURCE EXCERPT 10 · SQL LINES 17-27

```sql
    AND role = 'ADMIN'::public.app_role
  ) THEN
    RETURN TRUE;
  END IF;

  RETURN FALSE;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER STABLE;

-- 2. The Policy
CREATE POLICY "Admin Full Access Users" ON public.profiles FOR ALL USING
(public.is_admin());
```

5.3 Financial log permissions

Ordinary users cannot update or delete entries in `credits_transactions`; new ledger events record changes instead.

Insert	Allowed (via server-side triggers/functions).
Update / delete	Strictly Forbidden. No policy exists to allow modification of a transaction history row.

6. Identity lifecycle

Identity operations cover recovery, account deletion, sensitive changes, and email confirmation.

6.1 Password recovery

Implementation `src/actions/auth/forgot-password.ts`

Password recovery sends a Supabase recovery link back to the application. Use a configured, allowlisted application origin for that redirect; a request's `Origin` header alone is not a trust boundary.

SOURCE EXCERPT 11 · TYPESCRIPT LINES 1–9

```ts
// src/actions/auth/forgot-password.ts
export async function forgotPassword(email: string) {
    const origin = (await headers()).get('origin');

    // The link sent to email will redirect to /auth/reset-password
    const { data, error } = await supabase.auth.resetPasswordForEmail(email, {
        redirectTo: `${origin}/auth/reset-password`,
    });
}
```

6.2 Account deletion

Implementation `src/actions/auth/close-user-account.ts`

Account deletion verifies the current password through `verify_user_password` before calling `adminAuthClient.deleteUser`. Authorization and dependent-data cleanup must also be checked in the full action.

SOURCE EXCERPT 12 · TYPESCRIPT LINES 1–13

```ts
// src/actions/auth/close-user-account.ts
export async function closeUserAccount({ password }: { password: string }) {
    // 1. Re-Verification (Critical Security Step)
    const { data: isValidPassword } = await supabase.rpc('verify_user_password', {
        current_plain_password: password,
    });

    if (!isValidPassword) throw new Error('Incorrect Password');

    // 2. Hard Delete (Cascades via Foreign Keys)
    const supabaseAdmin = await createAdminClient();
    await supabaseAdmin.auth.admin.deleteUser(user.id, true);
}
```

6.3 High-assurance actions

Implementation `src/actions/auth/change-user-password.ts`

The documented account-deletion and password-change flows require fresh password verification through an RPC. Treat this as an additional check for sensitive actions, alongside the active session and the required assurance level.

SOURCE EXCERPT 13 · TYPESCRIPT LINES 1–10

```ts
// src/actions/auth/change-user-password.ts
export async function changeUserPassword(currentPassword, newPassword) {
    // 1. Re-Verify Old Password (SUDO Check)
    const { data: isValid } = await supabase.rpc('verify_user_password', {
        current_plain_password: currentPassword,
    }); // ... error handling

    // 2. Update to New Password
    await supabase.auth.updateUser({ password: newPassword });
}
```

6.4 Email change confirmation

Implementation `src/actions/auth/change-user-email.ts`

Email changes require confirmation. The source uses Next.js `after()` for deferred activity logging; check its completion behavior where an audit record is required.

SOURCE EXCERPT 14 · TYPESCRIPT LINES 1–11

```ts
// src/actions/auth/change-user-email.ts
export async function changeUserEmail(email: string) {
    // 1. Trigger Supabase "Double Opt-In" (Sends confirmation to NEW email)
    const { error } = await supabase.auth.updateUser({ email });

    // 2. Non-blocking Compliance Log
    // We use 'after' to ensure the user response is not delayed by the DB write
    after(async () => {
        await logUserActivity(user.id, 'changed_email', { newEmail: email });
    });
}
```

7. Security controls and excerpt limits

SECURITY CONTROL	IMPLEMENTATION	SOURCE LOCATION
Authentication	Supabase GoTrue (Email/Pass + OAuth)	`src/actions/auth/sign-in.ts`
MFA	TOTP (Time-based One-Time Password)	`src/actions/auth/multi-factor-authentication.ts`
Session Mgmt	HttpOnly, Secure, SameSite Cookies	`src/utils/supabase/middleware.ts`
Auth Callback	Code Exchange & Audit Logging	`src/app/auth/callback/route.ts`
Data Access	Row-Level Security (RLS)	`migrations/03_secure_rls.sql`
Lifecycle	SUDO Mode (RPC Verified Actions)	`src/actions/auth/change-user-password.ts`

EXCERPT BOUNDARY

The sign-in excerpt does not demonstrate a session-null check. Verify assurance-level enforcement in middleware, protected actions, and database policies. This handover documents the design and source examples; it does not certify the deployed security configuration.

PART I · THE ENGINE

Financial ledger

02

Transactions, credit integrity, payments, and refunds.

IN THIS SECTION

01	Ledger architecture	P. 023	02	Ledger schema	P. 023
03	Billing interface	P. 025	04	Concurrent credit deduction	P. 026
05	Payment ingestion	P. 028	06	Failure refunds	P. 030
07	Financial data flow	P. 032	08	Pricing rules	P. 033
09	Manual adjustments	P. 034	10	Financial reporting	P. 034
11	Balance refresh and checkout	P. 035	12	Integrity controls	P. 037
13	Operational configuration	P. 038			

Modules	`src/actions/billing/*` · `src/utils/supabase/credits.ts` · `migrations/03_secure_rls.sql`

1. Ledger architecture

The financial system uses a transaction ledger. Purchases, consumption, and refunds are recorded as events, with a cached balance maintained by a database trigger.

Typed ledger events record credit purchases, consumption, and refunds. The cached balance is a projection of that history, rather than an independent application-side counter.

The `sync_credits` trigger updates the cached balance when ledger entries are inserted. Its consistency depends on the trigger, permissions, transaction boundaries, and the set of transaction types it recognizes.

2. Ledger schema

Implementation

`supabase/migrations/00000000000002_core_schema.sql` & `00000000000003_secure_rls.sql`

`credits_transactions` stores the history of balance-changing events.

2.1 Tables and responsibilities

SOURCE EXCERPT 15 · SQL LINES 1–19

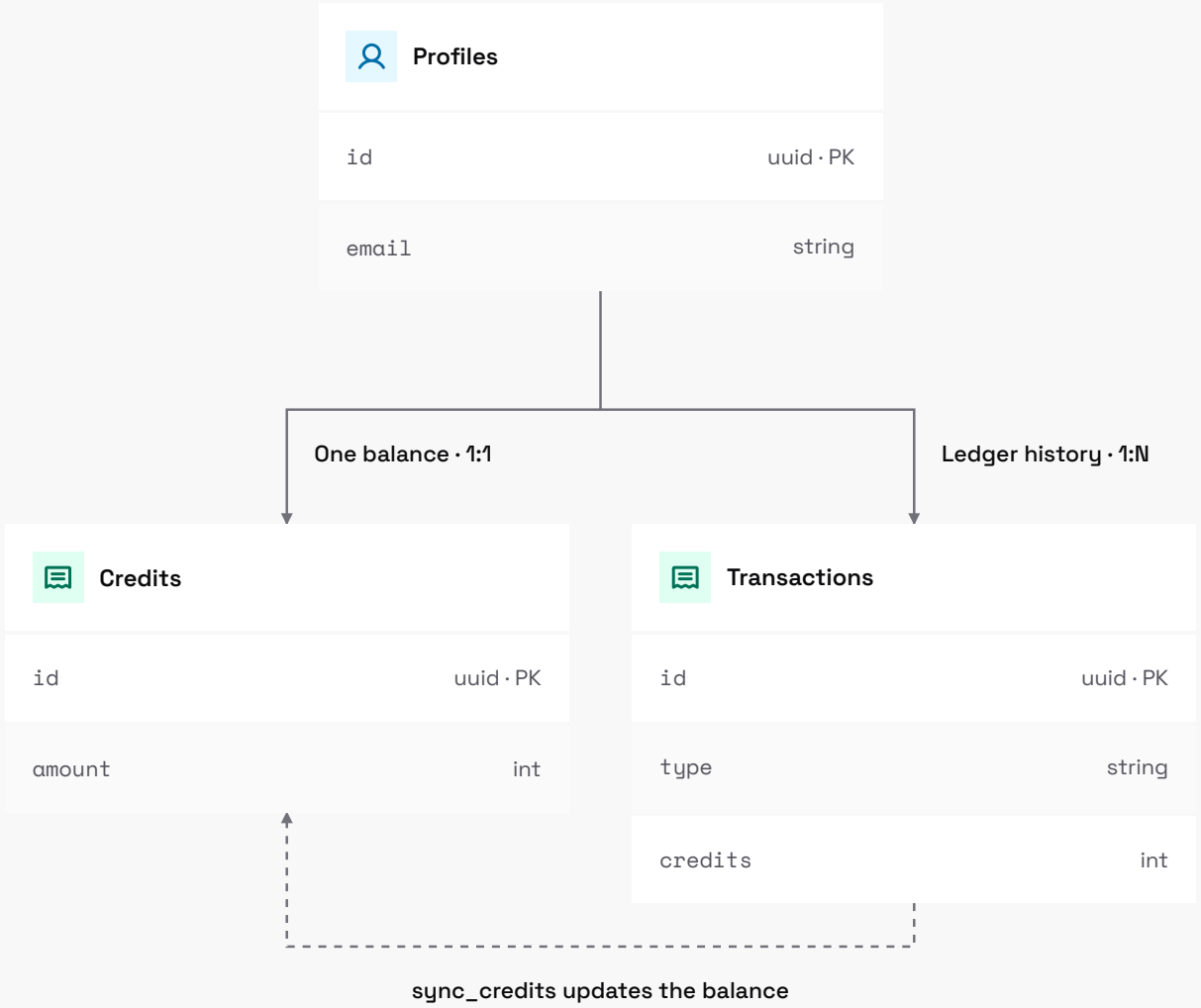
```sql
-- migrations/02_core_schema.sql

-- 1. The "State" (Cached Balance)
CREATE TABLE public.credits (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    profile_id UUID NOT NULL REFERENCES public.profiles(id) ON DELETE CASCADE UNIQUE,
    amount INTEGER NOT NULL DEFAULT 0 CHECK (amount >= 0),
    lifetime_purchased INTEGER NOT NULL DEFAULT 0,
    lifetime_used INTEGER NOT NULL DEFAULT 0,
    last_purchase_at TIMESTAMPTZ,
    last_usage_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- 2. The "Ledger" (Immutable History)
CREATE TABLE public.credits_transactions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    profile_id UUID NOT NULL REFERENCES public.profiles(id) ON DELETE CASCADE,
```

SOURCE EXCERPT 15 · SQL LINES 20–27

```sql
    translation_id UUID REFERENCES public.translations(id) ON DELETE SET NULL,
    credits INTEGER NOT NULL CHECK (credits > 0),
    type VARCHAR(50) NOT NULL, -- 'purchase', 'consumption', 'bonus', 'penalty', 'refund'
    description TEXT,
    details JSONB DEFAULT '{}'::jsonb,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

2.2 Entity relationships

04 Ledger entity relationships



amount is the current balance; credits is the transaction delta.
The source ERD lists purchase and consumption transaction types.

2.3 Balance synchronization trigger

The documented write path inserts into `credits_transactions`. The trigger below applies recognized transaction types to `credits.amount`.

SOURCE EXCERPT 16 · SQL LINES 1–18

```sql
-- migrations/03_secure_rls.sql

CREATE OR REPLACE FUNCTION sync_credits() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.type IN ('purchase', 'bonus', 'refund') THEN
        UPDATE public.credits SET amount = amount + NEW.credits WHERE profile_id = NEW.profile_id;
    ELSIF NEW.type IN ('penalty', 'consumption') THEN
        UPDATE public.credits SET amount = amount - NEW.credits WHERE profile_id = NEW.profile_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


-- The Enforcement
CREATE TRIGGER trigger_sync_credits
AFTER INSERT ON public.credits_transactions
FOR EACH ROW EXECUTE FUNCTION sync_credits();
```

3. Billing interface

Implementation `src/app/(protected)/account/*`

Two account views expose the ledger: credit purchase and billing history.

3.1 Credit purchase

Implementation `src/app/(protected)/account/credits/top-up/top-up-page.tsx`

The top-up view estimates the purchase and opens checkout:

Volume pricing	`0.10` -> `0.075` price tiers (Client-side estimation).
Stripe checkout	Creating Checkout Sessions (`/api/payment/checkout-session`).
Refresh balance	Using `react-query` to reflect new balances immediately after redirect.

3.2 Customer billing statement

Implementation `src/app/(protected)/account/billing`

The billing view presents transaction history and payment-document links.

Invoice history Renders the `credits_transactions` history.

PDF receipts Links directly to Stripe-hosted invoice PDFs.

3.3 Transaction history

The history view composes `TransactionHistoryPanel`, `CreditsTransactionTable`, and `FilterTransactionsForm`.

The transaction history provides pagination and filters for users reviewing a large ledger.

Infinite scroll

Load transaction history with `useInfiniteQuery` and an `IntersectionObserver` as the user scrolls.

Filters

Narrow the history by transaction type, amount, or date.

- **Type:** `consumption`, `purchase`, `bonus`, `penalty`, or `refund`.
- **Amount:** Minimum and maximum credit values.
- **Date:** A range selected in the calendar.

Transaction types

Color distinguishes transaction types, including green purchases and red consumption.

4. Concurrent credit deduction

Implementation `src/utils/supabase/credits.ts`

Concurrent requests must not spend the same balance twice. With 10 credits available, two simultaneous 10-credit requests must be serialized at the database boundary.

The shown PostgreSQL RPC locks the balance row while checking and deducting credits. Keep the deduction and ledger insertion within the same database transaction.

4.1 Deduction sequence

Implementation `deduct_credits_for_translation`

The deduction RPC checks available credits and inserts the consumption event. `sync_credits` applies the balance change inside the transaction.

THE TRANSACTION BLOCK

01 Lock the balance

Locks the user's `credits` row. Use `SELECT ... FOR UPDATE` to serialize access.

02 Check funds

Checks if `v_current_balance < p_credits_required`.

03 Record usage

Inserts a `consumption` record into `credits_transactions`.

04 Sync balance

The insert fires `sync_credits`, which updates the balance.

SOURCE EXCERPT 17 · SQL LINES 1–22

```sql
-- migrations/03_secure_rls.sql

CREATE OR REPLACE FUNCTION deduct_credits_for_translation(
  p_user_id UUID,
  p_translation_id UUID,
  p_credits_required INTEGER,
  p_document_name TEXT,
  p_page_count INTEGER
) RETURNS JSONB AS $$
DECLARE
  v_current_balance INTEGER;
  v_transaction_id UUID;
BEGIN
  -- 1. Security Check (Service Role or Own Usage)
  IF (auth.jwt() ->> 'role') <> 'service_role' AND auth.uid() <> p_user_id THEN
    RAISE EXCEPTION 'Unauthorized: You can only deduct your own credits.';
  END IF;

  -- 2. Lock & Check Balance
  SELECT amount INTO v_current_balance FROM public.credits WHERE profile_id = p_user_id
FOR UPDATE;

  IF NOT FOUND THEN RAISE EXCEPTION 'Credits record not found'; END IF;
```

SOURCE EXCERPT 17 · SQL LINES 23-40

```sql

  IF v_current_balance < p_credits_required THEN
    RETURN jsonb_build_object('success', false, 'error', 'Insufficient credits');
  END IF;

  -- 3. Insert Transaction (The Trigger will auto-update balance)
  INSERT INTO public.credits_transactions (
    profile_id, translation_id, credits, type, description, details
  ) VALUES (
    p_user_id, p_translation_id, p_credits_required, 'consumption',
    'Translation: ' || p_document_name, jsonb_build_object('page_count', p_page_count)
  )
  RETURNING id INTO v_transaction_id;

  -- 4. Return New Balance (Calculated Optimistically)
  RETURN jsonb_build_object('success', true, 'new_balance', v_current_balance -
p_credits_required, 'transaction_id', v_transaction_id);
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

5. Payment ingestion

Implementation

`src/actions/billing/stripe/server-only/events/handle-completed-checkout-session.ts`

The checkout interface starts payment; the webhook handler fulfills the purchase after verifying the provider event.

5.1 Webhook processing

01 Start checkout

Attach the user ID to the Stripe Checkout Session metadata.

02 Verify the event

Validate the webhook signature and handle the completed checkout event. Check payment status for the supported payment method.

03 Check prior processing

Look up the Stripe session ID. This is a duplicate-delivery check; concurrent processing also needs database enforcement.

04 Record the purchase

Insert a `purchase` ledger entry and check the write result. The `sync_credits` trigger updates the cached balance.

5.2 Webhook handler excerpt

SOURCE EXCERPT 18 · TYPESCRIPT LINES 1–24

```ts
// src/actions/billing/stripe/server-only/events/handle-completed-checkout-session.ts

export async function handleCompletedCheckoutSession(session: Stripe.Checkout.Session)
{
    const supabaseAdmin = await createAdminClient();

    // ... (Extraction of profileId and credits)

    // 1. Idempotency Check: Prevent double crediting
    const { data: existingTransaction } = await supabaseAdmin
        .from('credits_transactions')
        .select('id')
        .contains('details', { session_id: session.id })
        .maybeSingle();

    if (existingTransaction) {
        return logger.info(['Session already processed, skipping:', session.id]);
    }

    // ... (Formatting details, currency conversion)

    // 2. Insert into Ledger (Trigger Updates Balance)
    const { data, error } = await supabaseAdmin.from('credits_transactions').insert({
        credits: parseInt(credits),
        description: formattedDetails.description,
```

SOURCE EXCERPT 18 · TYPESCRIPT LINES 25-38

```
        details: {
            ...formattedDetails,
        },
        profile_id: profileId,
        type: 'purchase', // <--- Trigger adds credits for 'purchase'
    });

    if (error) {
        return logger.error(['Error updating credits:', error]);
    }

    // 3. Notify User
    await createNotification({ ... });
}
```

IMPLEMENTATION CHECK

The payment excerpt checks for an existing Stripe session before insertion. Concurrent delivery still needs a database uniqueness constraint or an equivalent atomic mechanism; that constraint is not shown here.

6. Failure refunds

Implementation `src/utils/supabase/credits.ts`

When asynchronous processing fails after charging credits, a compensating transaction records the refund.

Insert a new `refund` entry rather than removing the original consumption event.

SOURCE EXCERPT 19 · TYPESCRIPT LINES 1-17

```
// src/utils/supabase/credits.ts

export async function refundCreditsForFailedTranslation(
    supabase: SupabaseClient,
    params: { userId: string; translationId: string; creditsToRefund: number; reason: string },
): Promise<{ success: boolean; newBalance?: number; error?: string }> {
    // ... (Fetch current balance)

    // 1. Idempotency Check (Prevent looping refunds)
    const { data: existingRefund } = await supabase
        .from('credits_transactions')
        .select('id')
        .eq('translation_id', params.translationId)
        .eq('type', 'refund')
        .maybeSingle();

    if (existingRefund) return { success: true };
```

SOURCE EXCERPT 19 · TYPESCRIPT LINES 18–31

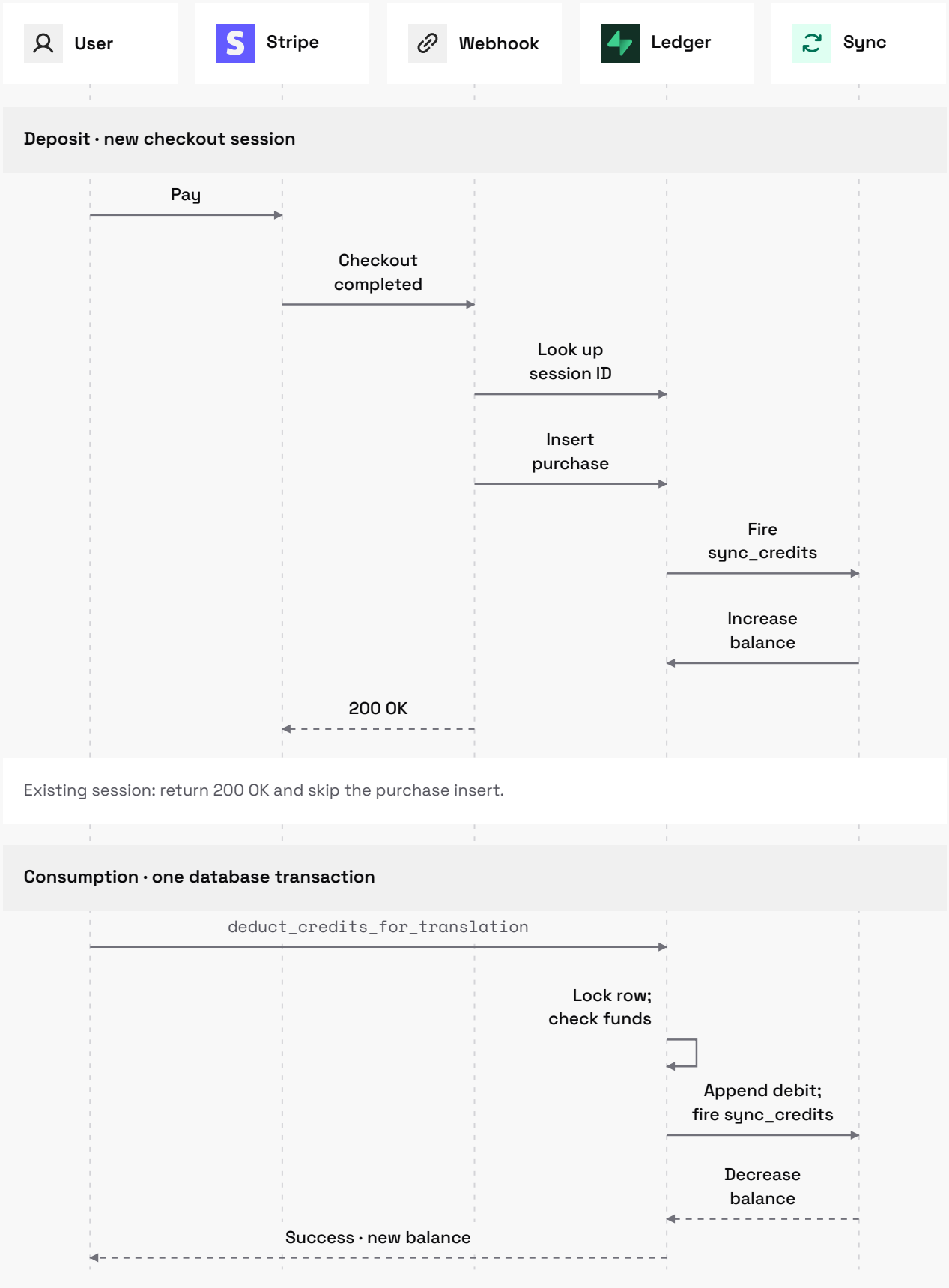
```
18
19      // 2. Log Refund Transaction
20      // NOTE: This insert triggers 'sync_credits', which ADDS the credits back.
21      const { error: transactionError } = await
    supabase.from('credits_transactions').insert({
22          profile_id: params.userId,
23          translation_id: params.translationId,
24          credits: params.creditsToRefund,
25          type: 'refund', // <--- Trigger adds credits for 'refund'
26          description: 'Credit refund for failed translation',
27          details: { reason: params.reason, refunded_at: new Date().toISOString() },
28      });
29
30      return { success: true };
31  }
```

IMPLEMENTATION CHECK

The refund excerpt returns success without checking `transactionError`. Confirm that the compensating entry was inserted, and enforce refund idempotency at the database boundary before reporting a restored balance.

7. Financial data flow

05 Financial data flow



8. Pricing rules

Implementation `src/constants/credits.ts`

The ledger stores credit events. The pricing rules determine how many credits a translation consumes and how credit packs are priced.

8.1 Credit calculation

Minimum charge

The shown calculation charges at least 3 credits per translation.

Page rate

Above that minimum, the shown function charges 1 credit per page.

Purchase tiers

Credit-pack pricing is configured separately from per-translation consumption.

SOURCE EXCERPT 20 · TYPESCRIPT LINES 1–19

```ts
// src/constants/credits.ts

export const PAGES_PER_CREDIT = 1;
export const MINIMUM_CREDITS_PER_TRANSLATION = 3;

/**
 * Pricing model: 1 credit per 1 page, minimum 3 credits.
 */
export const calculateCreditsRequired = (pageCount: number): number => {
    if (pageCount <= 0) throw new Error('Page count must be greater than 0');
    return Math.max(MINIMUM_CREDITS_PER_TRANSLATION, Math.ceil(pageCount / PAGES_PER_CREDIT));
};

// ... (Bulk Discount Logic)
export const calculateDiscount = (amount: number) => {
    if (amount >= 2000) return 60; // Whale (60% Off)
    if (amount >= 500) return 50; // Enterprise (50% Off)
    return 0;
};
```

9. Manual adjustments

Implementation `src/actions/admin/users/adjust-credits.ts`

Support grants and penalties use the ledger's normal write path. Administrative tools should not edit `credits.amount` directly.

9.1 Administrative action

The administrative action inserts a `bonus` or `penalty` transaction. The balance trigger processes that entry through the same ledger path as other recognized events.

SOURCE EXCERPT 21 · TYPESCRIPT LINES 1–23

```ts
// src/actions/admin/users/adjust-credits.ts

export async function adjustCredits({ userId, amount, type, description }) {
    // 1. Validation
    const { user: currentUser } = await validateUser({ requiredRole: 'ADMIN' });

    // 2. Insert Transaction (Trigger handles balance update)
    const { error } = await supabase.from('credits_transactions').insert({
        profile_id: userId,
        credits: amount,
        type: type, // 'bonus' adds, 'penalty' subtracts via Trigger
        description: description,
        details: { admin_id: currentUser.id }, // Audit who did it
    });

    // 3. Security Audit Log
    await supabase.from('audit_log_entries').insert({
        action: 'ADJUST_CREDITS',
        actor_id: currentUser.id,
        target_id: userId,
        changes: { amount, type, description },
    });
}
```

10. Financial reporting

Implementation `src/actions/admin/analytics/statistics/get-credits-burn-down-analytics.ts`

The transaction timestamps also support usage and purchase reporting.

10.1 Credit usage analysis

Aggregate transaction history to compare purchases and consumption over time.

SOURCE EXCERPT 22 · TYPESCRIPT LINES 1–21

```ts
// src/actions/admin/analytics/statistics/get-credits-burn-down-analytics.ts

export async function getCreditsBurnDownAnalytics({ startDate, endDate }) {
    // ... (Validation)

    const query = supabase
        .from('credits_transactions')
        .select('credits, type, created_at')
        .order('created_at', { ascending: false });

    if (startDate) query.gte('created_at', startDate.toISOString());

    // ... (Processing for Chart.js / Recharts)
    const transactionHistory = data.map((tx) => ({
        credits: tx.credits,
        type: tx.type, // 'consumption' vs 'purchase'
        time: new Date(tx.created_at).getTime(),
    }));

    return { data: transactionHistory };
}
```

11. Balance refresh and checkout

Implementation

`src/hooks/queries/auth/use-balance-data.ts` & `src/queries/tanstack/billing-query-options.ts`

The interface must refresh its cached balance after relevant events so users can see ledger changes.

11.1 Refresh policy

The documented balance view uses TanStack Query refresh events rather than a dedicated realtime subscription.

Cached balance

Treat the cached balance as stale immediately (`staleTime: 0`).

Refresh events

Refresh the balance when the user returns to the page, after a relevant mutation, and on mount.

- **Window focus:** Refetch when the user returns from another tab, such as Stripe checkout.
- **Invalidation:** Actions such as `adjustCredits` call `revalidatePath`.
- **Mount:** Fetch the current balance when the component mounts.

SOURCE EXCERPT 23 · TYPESCRIPT LINES 1–7

```ts
// src/queries/tanstack/billing-query-options.ts
export const balanceDataQueryOptions = (userId?: string) =>
    queryOptions({
        staleTime: 0,
        refetchOnWindowFocus: true, // Key for Stripe return flow
        queryFn: async () => { ... }
    });
```

11.2 Commerce configuration

The transaction's `details` JSONB stores currency and payment metadata.

MULTI-CURRENCY HANDLING

Base currency	System operates in `USD`.
Currency exchange	If a user pays in `EUR`, we calculate and store the `fx_rate` and `source_currency` directly in the ledger transaction.
Conversion fee	We explicitly track the 2% conversion markup for audit purposes.

Store the invoice and receipt references returned by Stripe with the transaction. These references help locate the original payment documents; their continued availability depends on the provider.

- `invoice_pdf`: Direct link to the tax invoice.
- `receipt_pdf`: Direct link to the payment receipt.

11.3 First-purchase state

The documented initial account has no free credit grant.

Create the balance

The documented `create_credits_on_profile_insert` trigger creates a `public.credits` row with `amount = 0`.

First purchase

A new account must obtain credits before starting paid translation work.

12. Integrity controls

CONTROL	IMPLEMENTATION	SOURCE OF TRUTH
Concurrent deduction	Row lock inside the deduction RPC	`deduct_credits_for_translation`
Balance projection	Ledger insertion invokes `sync_credits`	Database trigger
Transaction history	Append-only access policy for ordinary users	Ledger permissions
Webhook authenticity	Verify the Stripe event signature	Webhook handler
Duplicate deposits	Session lookup; database uniqueness also required	See the webhook implementation check
Credit settlement	Check and insert in one database transaction	Deduction RPC

13. Operational configuration

Provision these environment values and provider integrations for the billing flow.

13.1 Environment dependencies

Implementation `src/env.ts`

ENVIRONMENT VARIABLE	BOUNDARY	PURPOSE
`STRIPE_SECRET_KEY`	Server	Authenticates API calls to Stripe (Charges, Portal).
`STRIPE_WEBHOOK_SECRET`	Server	Verifies signature of incoming webhook events to prevent spoofing.
`NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY`	Client	Initializes Stripe Elements for the "Top-Up" UI.

13.2 Stripe integration

Stripe-hosted checkout and customer-portal pages are created through application API calls.

Checkout session

01 Start checkout

User clicks "Checkout" in `top-up-page.tsx`.

02 Create session

Creates a hosted Stripe Checkout page.

03 Return to billing

Redirects back to `/account/billing?success=true`.

Customer portal

01 Open portal

User clicks "Customer Portal" in `transaction-history-panel.tsx`.

02 Create session

`openCustomerPortal` server action -> `billing.portal.sessions.create`.

03 Self-service

Allows users to download receipts, manage saved cards, and view invoice history natively on Stripe.

Webhook endpoint and events

Endpoint

`/api/payment/webhook`

Webhook events

- `checkout.session.completed`: Credits user balance upon successful payment.
  - `checkout.session.async_payment_failed`: Handles failures for delayed payment methods (e.g., SEPA).
  - `customer.deleted`: Reserved for future cleanup logic (currently no-op).

PART I · THE ENGINE

Intelligence pipeline

03

Vision processing, orchestration, and document compilation.

IN THIS SECTION

01 Vision-first processing P. 041
02 Durable orchestration P. 043
03 Document translation engine P. 045
04 Responses API adapter P. 049
05 Failure recovery P. 049
06 Visual quality checks P. 051
07 Document compilation P. 052
08 Translation verification P. 053
09 Pipeline observability P. 054
10 Document types P. 057
11 Capacity and runtime P. 059
12 Processing cost estimates P. 060
13 Operational runbook P. 062
14 Architecture decisions P. 063
15 Implementation summary P. 065

Modules	`src/utils/ai/*` · `src/utils/trigger/*` · `src/config/ai.ts`

1. Vision-first processing

The translation path uses page images to interpret content and structure before rebuilding an editable document.

The pipeline uses page images to recover both text and layout. Its output is an editable document structure that can be reviewed and corrected before delivery.

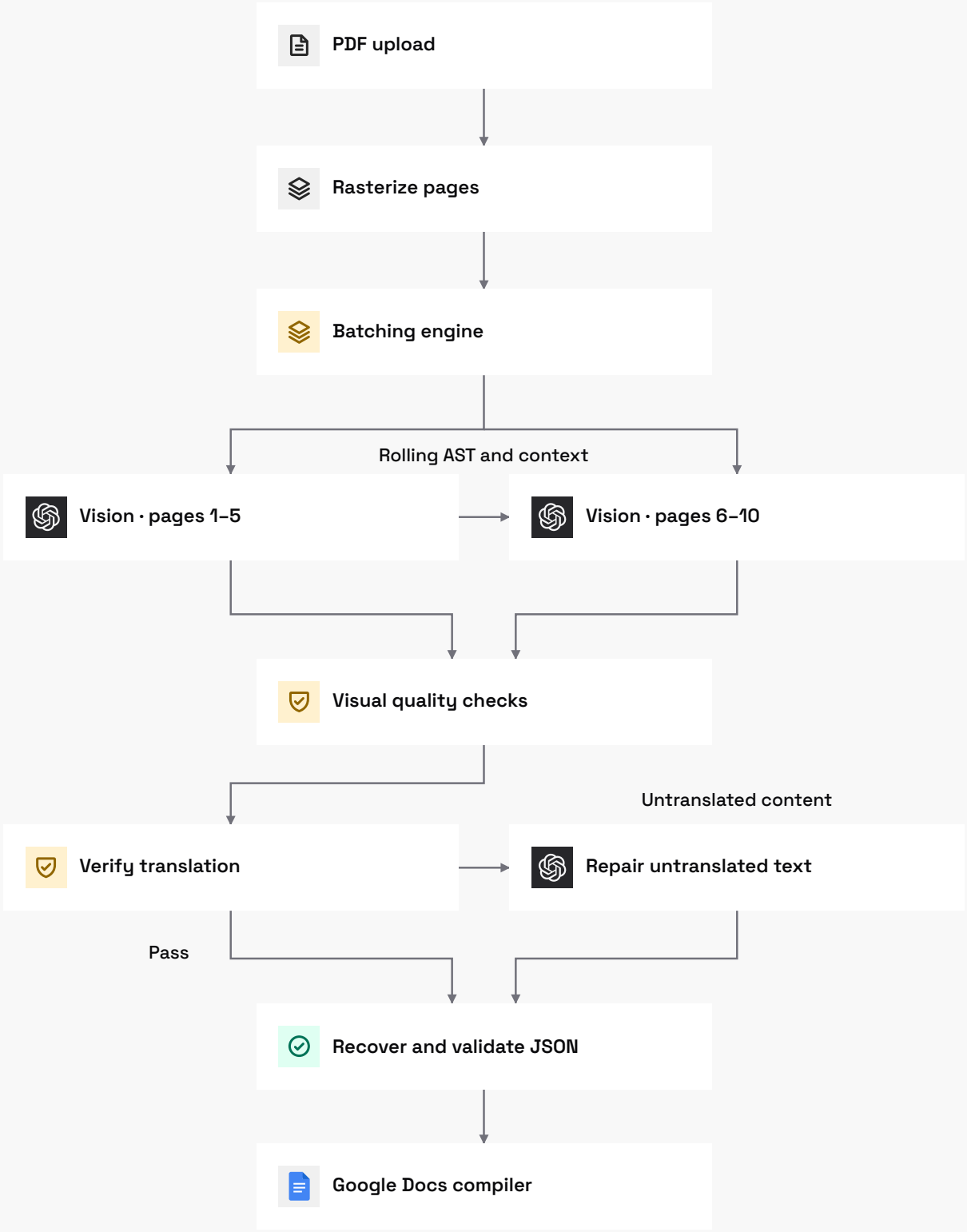
NOTE

Model Hierarchy:

- **Primary Intelligence (`gpt-5.2`)**: Handles the core Translation and recursive "Self-Correction" loops.
- **Speed Layer (`gpt-5-mini`)**: Handles metadata extraction, verification checks, and rapid "Repair" tasks to optimize latency and cost.

1.1 Pipeline architecture

06 Vision architecture



2. Durable orchestration

Implementation `src/utils/trigger/tasks/generate-docs-task.ts`

`visionTranslateDocumentTask` coordinates four weighted processing stages on Trigger.dev:

01 Prepare

0–15% · Download, normalize and capture image metadata.

02 Translate

15–65% · Translate with GPT-5.2 Vision.

03 Review

65–90% · Review and correct the output with GPT-5-mini.

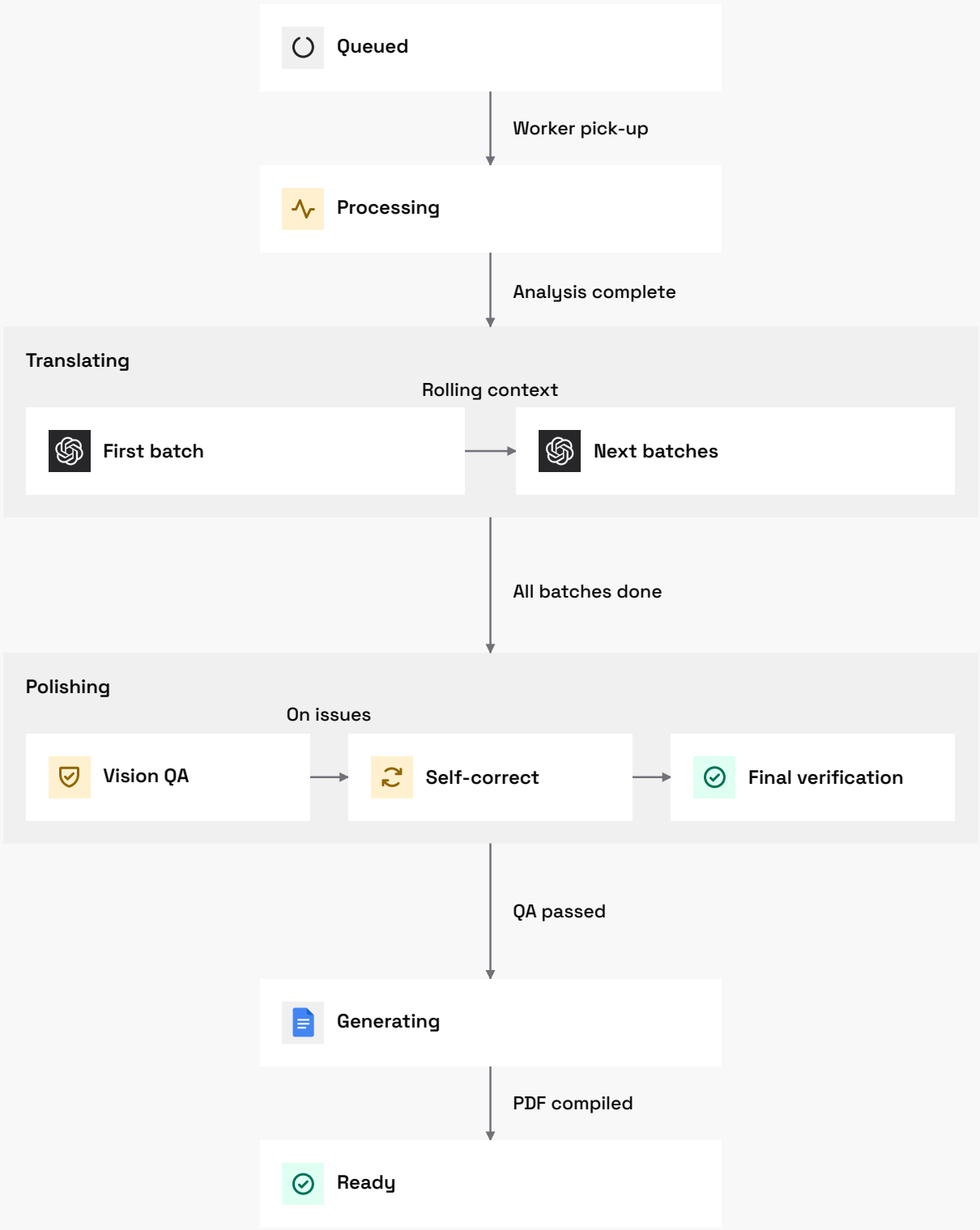
04 Generate

90–100% · Compile and export PDF / DOCX.

2.1 Lifecycle states

The lifecycle diagram reserves the valid states and transitions of the translation job.

07 Pipeline lifecycle



Lifecycle states drive progress updates while Trigger.dev runs the background workflow. The job can outlive a frontend HTTP request; its deadline, retry policy, and checkpoints are configured separately.

2.2 Execution pipeline

`generateDocsTask` sequences the documented processing stages and passes serialized state between them.

SOURCE EXCERPT 24 · TYPESCRIPT LINES 1–23

```ts
// src/utils/trigger/tasks/generate-docs-task.ts
export const generateDocsTask = task({
    id: 'generate-docs',
    retry: STANDARD_RETRY_CONFIG, // Exponential Backoff
    run: async (payload, { ctx }) => {
        // Step 1: Update Status
        await updateStatus('generating');

        // Step 2: VisionTranslate (The Recursive Loop)
        const result = await generateDocumentTask.triggerAndWait({
            translationId: payload.translationId
        });

        // Step 3: VisionQA (The Polishing Pass)
        const qaResult = await qaTask.triggerAndWait({
            document: result.document,
            originalImages: result.images
        });

        // Step 4: Finalize
        await finalizeTranslationTask.triggerAndWait({ ... });
    }
});
```

3. Document translation engine

Implementation `src/utils/ai/vision-document-translator.ts`

The translation module calls the model through `src/utils/ai/providers/openai-adapter.ts`.

3.1 Translation instructions

The system instructions define how to treat names, entities, legal terminology, and document structure. These are translation rules for review, not a certification of the output.

Identify content

”Does this text identify WHO or WHERE?” (If yes -> PROTECT/Do Not Translate).

Describe content

”Does this text describe WHAT something is?” (If yes -> TRANSLATE).

Legal entities

Explicit rules for Government Bodies, Academic Titles, and Legal Terminology.

3.2 Input checks and normalization

Before translation, normalize the input and collect the metadata used for batching.

01 Normalize input

Convert supported images and legacy PDFs to the pipeline’s standard PDF representation.

02 Name the file

A model pass reads the first page and proposes a descriptive filename.

03 Measure density

Estimate text density to select a batch size; see “Dynamic batch sizing”.

3.3 Contextual batching

Each batch receives context from earlier pages to reduce inconsistent terminology and broken document hierarchy:

Entity context

Pass names already encountered, such as people and organizations, to reduce inconsistent treatment between batches.

Terminology context

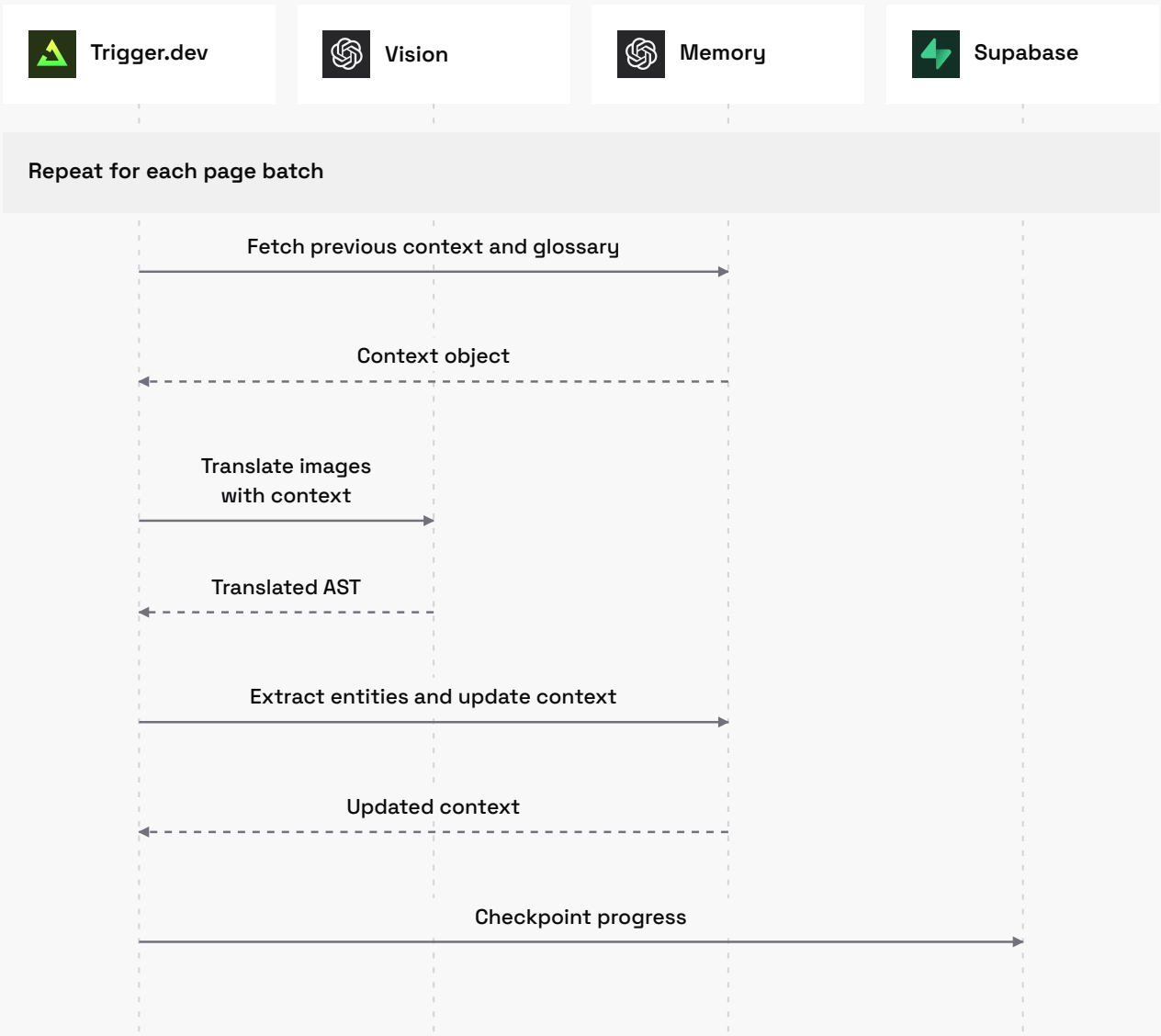
Pass the selected glossary and legal terms to support consistent wording.

Structural context

Pass preceding headings and document structure so later batches can continue the hierarchy.

This rolling context lasts for the task. It is not evidence of a separate retrieval index or persistent knowledge base.

08 Contextual batching



Vision: GPT-5.2. Memory: GPT-5-mini.
The next batch receives the updated context; the database retains progress.

SOURCE EXCERPT 25 · TYPESCRIPT LINES 1–8

```ts
// src/utils/ai/vision-document-translator.ts
// CRITICAL: Include context from previous batches to prevent inconsistencies
const batchInstruction = `
    CONTEXT FROM PREVIOUS BATCHES:
    - Entities found: ${previousContext.entities.join(', ')}
    - Terminology: ${JSON.stringify(previousContext.terminology)}
    - Recent Headings: ${previousContext.recentHeadings.join(' > ')}
`;
```

4. Responses API adapter

Implementation `src/utils/ai/providers/openai-adapter.ts`

The adapter uses `openai.responses.create`. The relevant controls are reasoning effort, output detail, and HTTP transport.

Reasoning effort

Select an effort supported by the chosen model. GPT-5.2 supports `none`, `low`, `medium`, `high`, and `xhigh`; use a lower effort only after evaluating the task.

Output detail

The API accepts `text: { verbosity: "low" | "medium" | "high" }`. Verbosity influences detail, not a hard output-token limit.

HTTP transport

The adapter configures an `undici.Agent`. Its request timeouts and idle keep-alive policy are separate from the worker's maximum task duration.

SOURCE EXCERPT 26 · TYPESCRIPT LINES 1–8

```ts
// src/config/ai.ts
export const AI_CONFIG = {
    model: 'gpt-5.2', // The Generative AI Model
    tasks: {
        translation: 'gpt-5.2', // High Intelligence
        metadata: 'gpt-5-mini', // High Speed
    },
};
```

5. Failure recovery

5.1 JSON recovery

The JSON recovery utility attempts to repair malformed output. Validate the recovered structure before accepting it; repair cannot recreate missing source content reliably.

01 Detects broken JSON tails.

02 Auto-closes open arrays/objects.

03 Salvages all valid translation nodes up to the cut-off point.

5.2 Checkpoints and retries

Before starting translation, the workflow checks `translation.details.translated_document` for a reusable checkpoint.

Resume checkpoint

If a usable `translation.details.translated_document` checkpoint exists, reuse it rather than repeating translation after a later export failure.

Cost boundary

Checkpoint reuse can avoid repeated inference. Verify checkpoint persistence, concurrent-run handling, and credit settlement before treating retries as cost-free.

5.3 Dynamic batch sizing

Implementation `batch-sizing.ts`

The pipeline varies batch size according to estimated document density.

Dense contracts	Smaller batches (2-3 pages) to prevent token overflow.
Sparse slides	Larger batches (10 pages) for speed.

5.4 Refund on processing failure

Implementation `src/utils/trigger/tasks/translation-task.ts`

A processing failure initiates a compensating credit transaction. Refund completion must be confirmed by the database write; catching an exception alone does not prove that the balance was restored.

SOURCE EXCERPT 27 · TYPESCRIPT LINES 1–5

```ts
// Automatic Refund Logic
await refundCreditsForFailedTranslation(supabase, {
    creditsToRefund: transaction.credits, // Exact amount consumed
    reason: errorMsg || 'Translation task failed',
});
```

Failure handling also sends a notification that links back to the affected translation.

Authenticated destination

Failure notifications link to `/app/translations/:id`. The route must enforce the identity and tenant controls described in System 1.

5.5 Schema validation

Implementation `schema-validation.ts`

Validate each JSON response against the Zod schema before passing it to later stages.

Validate schema

`parseAndValidateJSON(schema, output)`

Reject bad output

If the AI returns a valid JSON that misses a required field (e.g. `content_type`), the system throws a specific `SchemaValidationError` rather than polluting the database with corrupt data.

6. Visual quality checks

Implementation `src/utils/ai/vision-document-qa.ts`

A second model pass compares the source with the generated document and proposes corrections. Human review remains necessary for translation accuracy and legal use.

6.1 Compare source and output

The visual QA pass receives two inputs:

- The **Original Image** (The Ground Truth).
- The **Translated JSON** (The Candidate).

The QA pass compares source images with the generated document and proposes changes to layout and content. Treat its output as a review aid.

- "The original had a centered heading, but your JSON is left-aligned. **Fixing.**"
- "The original table had a merged cell `colspan=3`, you split it. **Fixing.**"

6.2 Progress reporting

The QA layer calls `report_progress` to send structured status messages to the frontend. These messages describe the current task; they are not a transcript of the model's internal reasoning.

ENGINEERING NOTE

"Checking table alignment on Page 3..." "Fixing header styles..." "Verifying entity protection..."

These activity messages give the user more useful feedback than an undifferentiated loading state.

6.3 Fallback when polishing fails

Implementation `src/utils/trigger/tasks/partials/polish-translation.ts`

An optional QA or polishing failure falls back to the available translated checkpoint. This fallback concerns output quality; it does not relax authentication or tenant controls.

SOURCE EXCERPT 28 · TYPESCRIPT LINES 1–7

```ts
catch (error) {
    logger.warn('Returning original document due to polish failure');
    return {
        code: 'POLISH_SKIPPED',
        data: { polishedDocument: payload.translatedDocument } // Return original
    };
}
```

If the optional polishing stage fails, the workflow can return the translated checkpoint with a warning. Earlier failures may still prevent document delivery.

7. Document compilation

Implementation `src/utils/gcp/google-docs/mapper.ts`

The `generate-document` stage compiles the shared document tree into the export representation.

7.1 AST-to-Docs mapping

The chosen compiler uses Google Docs for document layout and format export.

Document input

Accept the `NormalisedDocument` AST.

Compilation

`GoogleDocsMapper` translates nodes into `batchUpdate` requests.

- **Tables:** `insertTable` creates the grid; `updateTableCellStyle` applies cell rules and borders.
- **Headings:** `insertText` adds content; `updateParagraphStyle` applies a named heading style.
- **Images:** `insertInlineImage` inserts an image at a document index. Positioned-image handling is a separate concern.

7.2 Export formats

After compilation, the Drive export flow produces the required file formats. Completion and availability depend on the export request and storage write succeeding.

DOCX

Editable delivery for Microsoft Word.

PDF

Fixed-layout output for review and distribution. Export alone does not certify a translation.

HTML

Lightweight browser viewing.

7.3 Stored outputs and delivery

Download generated artifacts and write them to application storage for authenticated delivery. The user-facing download does not depend on access to the live Google Doc.

PDF output

`translations/:id/final.pdf`

DOCX output

`translations/:id/source.docx`

Delivery

Authenticated downloads from application storage do not require the recipient to use Google Docs.

8. Translation verification

Implementation `src/utils/ai/verification/verify-translation.ts`

Verification checks inspect the generated structure before delivery and flag content that may need repair.

8.1 Untranslated content detection

A language heuristic looks for blocks that appear to remain in the source language.

8.2 Repair pass

The documented threshold flags paragraphs with more than 5% untranslated content. `src/utils/ai/verification/fix-untranslated.ts` reprocesses the affected node.

8.3 Deliver with review warnings

Recoverable verification warnings do not abort the documented workflow. Preserve the result and make unresolved issues visible for review.

Return editable output

When a recoverable QA issue remains, return the available document for review rather than discarding completed work.

Expose warnings

Record unresolved checks in the audit data and show a clear review warning with the delivered result. This is the intended policy; confirm the actual user-facing warning in the application.

9. Pipeline observability

ENGINEERING NOTE

User Requirement:

"I want to verify the system without reading code or grepping log files."

Trigger.dev traces expose processing phases and events in an operator interface.

9.1 Progress telemetry

The documented progress design publishes task metadata for a live frontend view. System 4 separately describes the Supabase translation subscription.

Run tags

Filter runs by user, translation, page count, and language.

Example: `user:usr_123`, `id:trans_456`, `pages:10+`, `lang:en->es`.

Use: Find failed Spanish translations with more than ten pages.

Live progress

Stream processing state through the v4 Realtime SDK.

Example: `phase: translating`, `progress: 45%`, `current_operation: Processing Batch 2/5`.

Use: Update the frontend progress display.

Timing spans

Measure individual operations to locate slow stages.

Example: `vision-translation` (35s), `generate-docx` (1.2s), `qa-check` (5s).

Use: Investigate a three-minute run by opening its spans; an `upload-images` span may account for 2.5 minutes.

Event logs

Attach structured fields to events.

Example: `[QA] Review complete | fixes=3, confidence=98%`.

9.2 Logging implementation

Implementation `logging.ts`

Use stable phase names so logs from different processing stages can be compared.

SOURCE EXCERPT 29 · TYPESCRIPT LINES 1–10

```ts
// src/utils/trigger/logging.ts
export type LogPhase = 'INIT' | 'OCR' | 'TRANSLATE' | 'POLISH' | 'ERROR';

export function logPhase(phase: LogPhase, message: string, data?: object) {
    // Automatically attaches metrics to the span
    logger.info(`[${phase}] ${message}`, data);
}

// Result in Dashboard:
// [TRANSLATE] Batch 1 complete | pages=5, tokens=4500, duration=12s
```

9.3 Audit data

Implementation `src/types/ai-audit.ts`

A `QAStageAudit` object records the QA fields described below in a persistent JSONB column. This supports investigation; the excerpt does not establish complete decision capture or a legal compliance guarantee.

SOURCE EXCERPT 30 · TYPESCRIPT LINES 1–22

```typescript
// From src/types/ai-audit.ts
export type AIAuditData = {
    schema_version: 1;
    pipeline: {
        total_processing_ms: number;
        stages_completed: ('processing' | 'translation' | 'qa')[];
        // Critical for debugging: What settings were active?
        config_snapshot: {
            AI_MODEL_TRANSLATION: string; // e.g. "gpt-5.2"
            preset_translation: {
                reasoning: 'high' | 'medium';
                verbosity: 'normal' | 'verbose';
            };
        };
    };
    // Detailed breakdown of QA fixes
    qa: {
        issues_fixed: number;
        issue_breakdown: { alignment: number; table: number; entity: number };
        confidence: number;
    };
};
```

9.4 Input provenance

Processing metadata records the input path, processing actor, and upload time:

Source path

`originalFilePath` identifies the uploaded input in the user's storage area.

Processor

`uploadedBy` records the processing actor, shown as `Document AI Processor`.

Upload time

`uploadedAt` stores an ISO timestamp. Together these fields support tracing the processed input.

9.5 Progress history

`progress_history` stores the documented rolling window of 50 state changes:

Timestamp

When the progress event was recorded.

Completion

The reported percentage at that point in the job.

Activity

A user-facing label such as “Analyzing table structure”.

State transition

The change in job state. This history helps reconstruct the processing sequence.

10. Document types

10.1 Normalized document tree

`NormalisedDocument` is the recursive document tree shared by translation, editing, and compilation. Nodes represent content and layout independently of the export format.

Source `src/types/document/normalised-document.d.ts`

SOURCE EXCERPT 31 · TYPESCRIPT LINES 1–17

```ts
/**
 * The root document interface representing the entire document.
 * Contains metadata and content nodes.
 */
export type NormalisedDocument = {
    type: 'doc';
    content: ContentNode[];
    translatedContent?: ContentNode[];
    documentType: string;
    detectedLanguages?: DetectedLanguage[];
    translatedLanguage?: string;
    metadata?: any;
    enhancedContent?: any[];
};

/** Discriminated union of all content node types */
export type ContentNode =
```

SOURCE EXCERPT 31 · TYPESCRIPT LINES 18–35

```
      | TextNode
      | ParagraphNode
      | HeadingNode
      | ListNode
      | ListItemNode
      | TableNode
      | TableRowNode
      | TableCellNode
      | TableHeaderNode
      | ImageNode
      | KeyValuePairNode
      | HardBreakNode
      | ThematicBreakNode
      | BlockQuoteNode
      | FootnoteNode
      | SpacerNode
      | DescriptionListNode
      | DescriptionItemNode;
```

Refinement schema

Structured Outputs constrains the response shape, and Zod validates it against the refinement schema. Structural validity does not establish translation accuracy. Implementation: `src/schemas/document/refinement-schema.ts`.

10.2 Branding and PDF types

Typed branding settings describe watermarks, stamps, and page layout for generation.

Source `src/types/document/branding-settings.ts`

SOURCE EXCERPT 32 · TYPESCRIPT LINES 1–15

```
export type DocumentBrandingSettings = {
    enabled: boolean;
    logoUrl: string | null;
    companyName: string | null;
    headerText: string | null;
    footerText: string | null;
    showCertificationStamp: boolean;
    certificationNumber: string | null;
    certificationStampUrl: string | null;
    certificationStampPlacement: 'page_bottom_right_margin' | 'page_bottom_right' | 'footer_bottom_right';
    certificationStampSize: 'sm' | 'md' | 'lg';
    showPageNumbers: boolean;
    watermarkText: string | null;
    watermarkPreset: 'light' | 'dark' | 'off';
};
```

11. Capacity and runtime

11.1 Page-count limit

The documented operational limit is 500 pages. The worker configuration permits a task duration of up to 24 hours; neither setting guarantees that a job will complete.

DOCUMENT	PAGES	EST. DURATION	DENSITY
Certificate (Simple)	1-2	2-5 Minutes	Low
Death Certificate (Mixed)	1-3	6-15 Minutes	Medium
Corporate Report (Dense)	50+	40-50 Minutes	High
Legal Discovery (Massive)	500	2-4 Hours	High

CAPACITY ASSUMPTION

Page count, image detail, batching, retries, and provider response time all affect duration. Measure representative documents before setting an operational commitment.

11.2 Long-running execution

As described in ADR-002, translation runs as a durable Trigger.dev v4 job with its own processing deadline.

Task budget

The source configures `maxDuration: 86400` for translation tasks: a maximum of 24 hours. The four-to-five-hour workload estimate is an assumption, not a measured service commitment.

Connection budget

The source mentions a 24-hour idle keep-alive value here and a two-hour value in the infrastructure excerpt. These examples are inconsistent. Verify deployed client settings separately; task duration, response timeouts, and idle keep-alive are different controls.

Checkpoint retry

After a worker failure, resume from the stored `translated_document` checkpoint where the implementation supports it. Check persistence and retry behavior before relying on avoided inference costs.

12. Processing cost estimates

Image-based translation consumes vision input, text context, and output tokens. The estimate below separates these cost drivers.

12.1 Vision input cost

The source budget separates image input from text context and output. Higher image detail can help with small text in stamps and footnotes, at additional token cost.

Image-token calculation

The calculation below reproduces the manuscript's image-token assumption. It is not a verified GPT-5.2 billing formula; confirm model-specific image accounting before using it for a quote.

Base tokens

85 tokens

Image tiles

~6 tiles (512x512px chunks) × 170 tokens = 1,020 tokens

Total input

~1,105 tokens / page (Just to see the page)

Per-page estimate

COMPONENT	TOKENS	EST. COST	COST DRIVER
Vision input	1,105	$0.0055	Page image
Text context	~1,000	$0.0050	Previous pages and system prompt
Translation output	~500	$0.0075	$15 per 1M output tokens
Total	~2,605	~$0.0180	About 4× text-only ($0.004)

COST TRADEOFF

Pay for structure.

The source estimate prices a vision page at roughly $0.02. The intended tradeoff is less manual layout repair; actual savings depend on document complexity and the review required.

12.2 Complexity assumptions

The source uses these illustrative factors to estimate how document density changes processing cost:

Simple letter

1.0× baseline processing estimate.

Legal contract

2.2× estimated processing cost for dense text.

Financial table

3.0× estimated processing cost for dense numeric content. These assumptions are separate from the user credit formula.

13. Operational runbook

Use this sequence to investigate translation runs and choose a recovery action.

13.1 Investigate a stalled job

Example symptom: a user reports the same 15% progress value for more than an hour.

01 Locate the run

Open the Trigger.dev dashboard and find the `vision-translate-document` run for the affected translation.

02 Inspect evidence

For a queued run, check concurrency and capacity. For a running job, inspect recent logs, progress, and checkpoints. Cancel only after confirming that it is stalled or cannot recover.

03 Retry deliberately

Use the documented `admin_retry_translation(id)` RPC after checking the previous run and checkpoint. Confirm the new state and avoid two active runs for the same translation.

13.2 Error responses

ERROR PATTERN	RESPONSE	OPERATOR ACTION
`rate_limit_exceeded`	Transient Retry after 1s, 2s, 4s.	No intervention; automatic retry.
`context_length_exceeded`	Permanent Job fails.	Reduce `maxPagesPerBatch` in configuration.
`invalid_image_data`	Permanent Job fails.	Check for a corrupt or password-protected PDF.

14. Architecture decisions

The requirements, alternatives, and tradeoffs behind three core architecture decisions.

The estimates and platform limits reflect the comparison criteria. For a new deployment, measure runtime on representative documents and check current provider limits.

14.1 Generative vision

ADR-001 Accepted

REQUIREMENT

Translate the text while preserving the source document's merged cells, borderless tables, and signature placement. The evaluation compared three approaches.

AWS Textract — Rejected

Capability

Low-cost OCR; the recorded estimate was $1.50 per 1,000 pages.

Constraint

The evaluation found language coverage unsuitable for the Arabic, Chinese, and mixed-script documents in scope.

Google Document AI — Rejected

Capability

Fast OCR with broad language support.

Constraint

The evaluated output required fragile reconstruction of merged cells, borders, and semantic structure from text and coordinates.

GPT-5.2 Vision — Selected

Capability

Uses visual context to interpret document structure alongside text.

Tradeoff

The recorded estimate was about 45 seconds and $0.02 per page. The design accepts higher compute cost to reduce manual layout repair.

DECISION

Use generative vision for ingestion. Accept higher inference cost and latency to preserve document structure and reduce manual layout repair.

14.2 Durable execution

ADR-002 Accepted

WORKLOAD BUDGET

A 500-page job exceeds a single short-lived function invocation.

500	×	4s	=	33m 20s
pages		per page		serial execution

Illustrative calculation from this decision's workload assumption. End-to-end duration also depends on batching, retries, and document complexity.

EXECUTION PLATFORM	RECORDED LIMIT	IMPLICATION
Vercel functions	60s or 900s	Split execution An Inngest workflow on Vercel would split long jobs across invocations and carry state between them.
AWS Lambda	15 minutes	Split execution The recorded invocation limit is below the 33-minute serial workload.
Trigger.dev v4	Up to 24h	Selected A persistent container accommodates the job duration, with durable execution and tracing.

Limits recorded for this architecture decision. Inngest orchestrates the workflow; invocation limits belong to the function host.

DECISION

Run background translation in Trigger.dev v4. A durable job keeps the translation loop and its traces in one run, avoiding recursive function invocations.

14.3 Document compilation

ADR-003 Accepted

REQUIREMENT

Produce editable DOCX files and fixed-layout PDF records from the same document representation.

APPROACH	EVALUATION
Browser export	The source evaluation recorded styling instability and memory issues on documents above 500 pages.
Google Docs API	Use a document intermediate representation for layout, pagination, and multi-format export.

DECISION

Use Google Docs API as the document compiler. Accept export latency in exchange for centralized layout and pagination across PDF, DOCX, and EPUB output.

15. Implementation summary

CAPABILITY	CONVENTIONAL APPROACH	VERTO IMPLEMENTATION
Ingestion	OCR extraction	Generative vision with GPT-5.2
Translation	Text-only translation	Reasoning with document context
Quality checks	Human review	Vision comparison and correction
Progress	Generic processing status	Tool-level progress streaming
Audit	No decision log	Granular decision logging

PART II · THE EXPERIENCE

User workspace

04

The editor, source viewer, client records, and realtime state.

IN THIS SECTION

01 Workspace responsibilities P. 067
02 Translation editor P. 068
03 Source document viewer P. 076
04 Client records P. 079
05 Realtime synchronization P. 080
06 Workspace resilience P. 083
07 Workspace technology P. 084
08 Performance strategies P. 084
09 Editor decisions P. 085
10 User credit consumption P. 086

Modules	`src/components/translations/TranslationEditor/*` · `src/app/(protected)/app/customers`

1. Workspace responsibilities

The browser workspace combines translated content, the original source, client records, and editing controls.

It supports two review responsibilities:

Document layout

How to edit complex, layout-preserved documents without breaking the visual hierarchy?

Agency workflow

How to allow our users (translators) to manage their own clients within our platform?

1.1 Core editor component

Implementation `TranslationEditor`

The React editor receives `initialContent` as JSON and emits `onChange` events.

SOURCE EXCERPT 33 · TYPESCRIPT LINES 1–11

```ts
// src/components/translations/TranslationEditor/translation-editor.tsx
type TranslationEditorProps = {
    initialContent: JSONContent | null;
    onChange: (content: JSONContent) => void;
    editable?: boolean;
    originalContent?: JSONContent | null; // For reference/diffing
    onReset?: () => void;
    isSaving?: boolean; // Triggers footer badge
    lastSaveTime?: Date | null; // "Saved at 10:42 AM"
    hasUnsavedChanges?: boolean; // Controls "Save" button state
};
```

2. Translation editor

Implementation `src/components/translations/TranslationEditor/translation-editor.tsx`

Tiptap provides a headless editor built on ProseMirror's schema. Verto extends that schema to edit the document structures produced by the translation pipeline.

EDITING CONTRACT

Document model

Tiptap edits a JSON document tree governed by ProseMirror's schema. Content attributes must survive parsing and serialization.

Custom structures

Extensions represent description lists, footnotes, signature blocks, and tables in the editor's document model.

Toolbar state

Selectors subscribe toolbar controls to the editor state they use. Rendering behavior must be profiled in the full component tree.

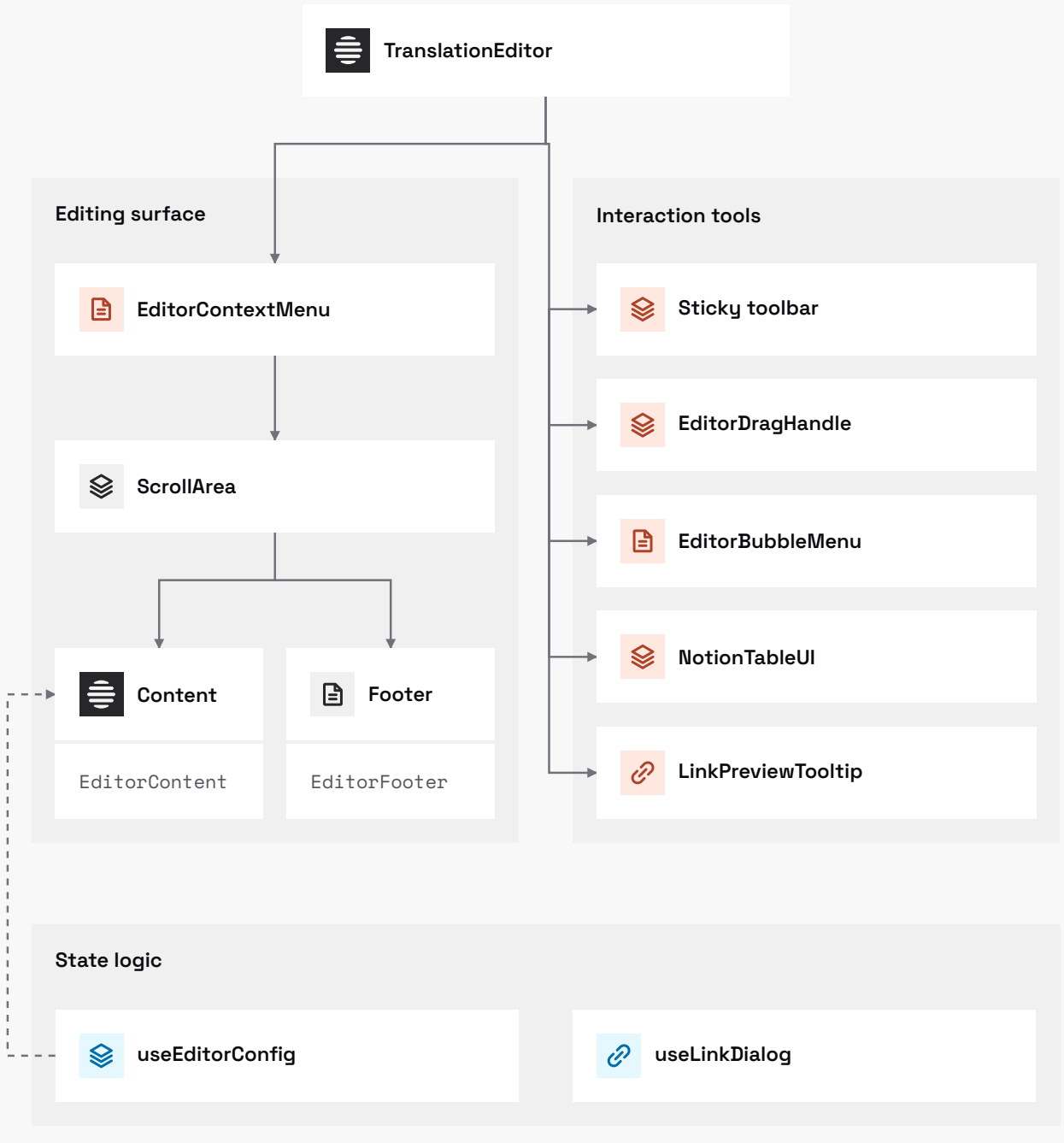
Regeneration

The documented regeneration flow applies saved user edits, then recompiles the document without rerunning translation.

2.1 Component composition

`TranslationEditor` composes specialized components for document editing and review.

09 Editor composition



Solid connectors show component ownership. Dashed connectors show configuration.

Editor config

`useEditorConfig` memoizes extensions and sanitizes input/output JSON to prevent attribute loss, including rows disappearing during `prosemirror` hydration.

Table controls

`NotionTableUI` attaches drag handles and “Add Column/Row” controls to tables, resolving the standard `contenteditable` UI limitations.

Loading state

`EditorSkeleton` uses structural clone placeholders to prevent layout shift.

2.2 Document schema extensions

Verto needs domain-specific structures such as signature blocks and description lists. Custom schema extensions represent those structures and connect them to the editor’s rendering and serialization.

Structure

`DescriptionList` (DL/DT/DD), `DescriptionItem`, `Footnote`

Formatting

`HeadingExtended`, `ParagraphExtended`, `Spacer`, `ThematicBreak`

Tables

`TableExtended`, `TableHeaderExtended`, `TableCellExtended`

Decorators

`LinkPreviewTooltip` (Hover cards for hyperlinks)

Implementation `src/components/translations/TranslationEditor/extensions/custom/*`

EXTENSION	PURPOSE	IMPLEMENTATION DETAIL
`TableExtended`	Handles "Invisible Tables"	Adds `kvp-table` class for borderless rendering and enables drag-handle detection on tables.
`DescriptionList`	Key-Value Pairs	Renders "Name: John Doe" structures not as text, but as semantic `<dl><dt><dd>` pairs to preserve alignment.
`ThematicBreak`	Page Separation	Renders a visual "Page Break" line so the user knows exactly where PDF pages end.
`Footnote`	Academic Citations	Manages superscripts and footer references, preserving legal document integrity.
`Spacer`	Visual Layout	Adds adjustable vertical whitespace blocks for precise document formatting.
`HeadingExtended`	Hierarchy Control	Custom heading nodes that enforce document structure (H1-H6) and allow deep linking.

2.3 Editor state

Implementation `src/components/translations/TranslationEditor/hooks/useToolbarState.ts`

`useEditorState` selects the editor state needed by the toolbar:

SOURCE EXCERPT 34 · TYPESCRIPT LINES 1–24

```ts
export type ToolbarState = {
    // Text formatting
    isBold: boolean;
    isItalic: boolean;
    isUnderline: boolean;
    isStrike: boolean;
    // ...
    // Command capabilities (Can I undo? Can I toggle bold?)
    canUndo: boolean;
    canRedo: boolean;
    canToggleBold: boolean;
};

// Only re-renders when "Toolbar-Relevant" state changes
export function useToolbarState(editor: Editor | null): ToolbarState | null {
    return useEditorState({
        editor,
        selector: ({ editor: e }): ToolbarState | null => ({
            isBold: e.isActive('bold'),
            canUndo: e.can().chain().focus().undo().run(),
            // ... mapped state
        }),
    });
}
```

The selector narrows the toolbar's subscription to the state it uses. Re-render behavior also depends on selector equality and the surrounding component tree; profile the full editor before claiming a performance result.

2.4 Regeneration flow

Implementation `src/actions/services/translations/regenerate-document.ts`

The documented editing flow keeps user changes distinct from the generated document and passes them into regeneration.

01 User edits

Store JSON Patch edits in `translations.details!.user_edits`.

02 Regeneration

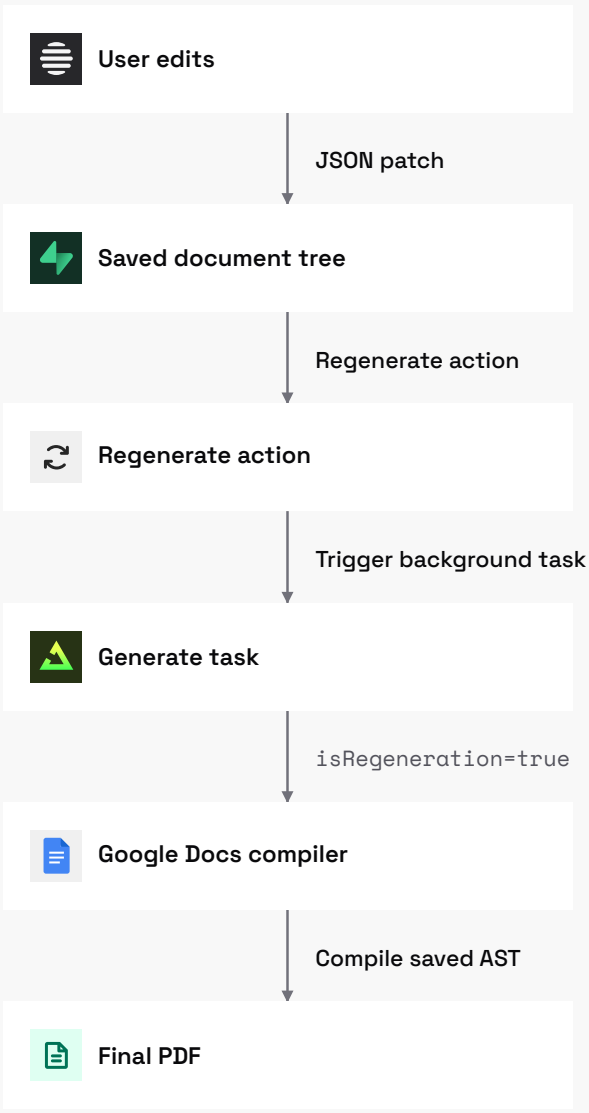
“Apply Changes” calls `regenerateDocument({ translationId })`.

03 Compilation

Run `generateDocumentTask` with `{ isRegeneration: true }`.

- Skip the AI translation stages.
- Rebuild the document from JSON through Google Docs to PDF.
- Retain document-generation compute costs while avoiding another translation pass.

10 Regeneration loop



Regeneration compiles the saved document tree.
The regeneration flag selects this path instead of a new vision translation.

2.5 Brand configuration

Implementation `src/components/branding/branding-editor.tsx`

The branding editor configures headers, footers, watermarks, and stamps separately from the document body.

Component architecture

Its reduced Tiptap schema permits bold, italic, and links through a custom `BubbleMenu`, keeping document-level blocks outside this field.

SOURCE EXCERPT 35 · TYPESCRIPT LINES 1-12

```ts
// src/components/branding/branding-editor.tsx
// A minimal, constraint-based editor for headers/footers
export function BrandingEditor({ content, onChange }: BrandingEditorProps) {
    const editor = useEditor({
        extensions: [
            StarterKit.configure({ heading: false, bulletList: false, ... }), // Disable block nodes
            LinkIcon, // Enable hyperlinks
        ],
        // ...
    });
    // Output: Returns a clean ProseMirrorJSON object
}
```

Pipeline integration

The branding UI produces the `DocumentBrandingSettings` consumed by the pipeline.

01 Edit branding

Header/Footer in `BrandingEditor`.

02 Save settings

JSON stored in `translations.branding_settings`.

03 Compile output

The pipeline's `GoogleDocsMapper` reads this JSON to inject headers/footers into the final PDF.

2.6 Editor interactions

Implementation

`src/components/translations/TranslationEditor/extensions/table-keyboard-shortcuts.ts`

Custom keyboard handlers support common table and content-editing actions:

INTERACTION	SHORTCUT	IMPLEMENTATION
Table Navigation	`Tab` / `Shift+Tab`	Overrides default focus behavior to move cursor between cells (Excel-style).
Markdown Paste	`Ctrl+V`	Detects Markdown syntax on paste (e.g. `**bold**`, `## Header`) and auto-formats it via `PasteMarkdown` extension.
Smart Links	`Cursor`	`LinkPreviewTooltip` appears when the caret is placed inside a link, allowing "Edit/Remove" without breaking flow.

3. Source document viewer

Implementation `src/components/translations/pdf-viewer.tsx`

The original PDF remains the review reference. The source viewer virtualizes pages to limit mounted canvases and browser memory during long-document review.

3.1 Page virtualization

`react-window` and `VariableSizeList` mount the visible page range rather than every page canvas.

Rendering cost

Rendering every page at once creates many canvas elements and increases memory pressure.

Virtual pages

Render the pages near the viewport.

- **Dimensions:** Fetch page aspect ratios in the background.
- **Visible pages:** Keep approximately two pages rendered at a time.
- **Fast scrolling:** Show skeletons until scrolling stops.

3.2 Zoom rendering

Zoom combines an immediate scaled preview with a replacement render at the requested resolution:

01 Preview the zoom

Apply a CSS `scale()` immediately to provide feedback during the gesture.

02 Render at resolution

The documented delay is 200ms before rendering the PDF at the new resolution.

03 Swap the canvas

Keep the previous image visible until its replacement is ready. Verify continuity and frame rate on representative documents.

3.3 Image viewer

Implementation `src/components/translations/image-viewer.tsx`

For scanned JPEG or PNG inputs, `react-zoom-pan-pinch` provides image zoom and pan controls.

Rotate and flip

Supports 360° rotation and H/V flipping (critical for poorly scanned documents).

EXIF orientation

Automatically respects the `imageOrientation: 'from-image'` tag to prevent upside-down loads.

Keyboard controls

`Alt+Shift+R` to rotate, `Cmd +/-` to zoom, mimicking native macOS Preview behavior.

3.4 Review layout

Implementation `src/components/translations/review-layout.tsx` & `document-viewer.tsx`

`DocumentViewer` selects the viewer for the source format:

01 Detect file type

Checks file extension (and MIME type fallback).

02 Select viewer

Renders `<PdfViewer />` or `<ImageViewer />` dynamically.

03 Preserve layout

Both viewers share the same `isCollapsed` state managed by the parent `ReviewLayout`.

The review layout combines panel resizing, draft state, shortcuts, and narrow-screen behavior:

Split panels

`react-resizable-panels` provides the source and editor split.

Saved layout

Persist the panel ratio in `localStorage` under `review-panel-sizes`.

Draft recovery

The documented `navigator.sendBeacon` attempt is best-effort. Use an acknowledged save as the confirmation that a draft reached the server.

Keyboard actions

- `Cmd+S` — save draft.
- `Cmd+Enter` — approve translation.

Narrow screens

Below the documented 1000px breakpoint, switch from split panels to a tabbed view.

4. Client records

Implementation `src/app/(protected)/app/customers` & `src/actions/services/customers`

Client records let a translation agency group work by its own end-clients.

4.1 Data model

The authenticated account owns the client records and translations within its tenant boundary.

Customer record

Represents an Agency's client.

```
// Schema Reference
type Customer = {
    id: string;
    user_id: string; // The Agency (Verto User)
    name: string;
    email: string;
    // ...
};
```

Translation link

A `translation` row can be linked to a `customer_id`. This allows the Agency to filter "All translations for Client X."

4.2 Client operations

Client changes

`add-customer.ts` and `update-customer.ts` perform server-side writes under the user's RLS context. Verify transaction and error handling in those actions before describing them as atomic.

Translation assignment

`assign-customer-modal` links a completed translation to a client record.

5. Realtime synchronization

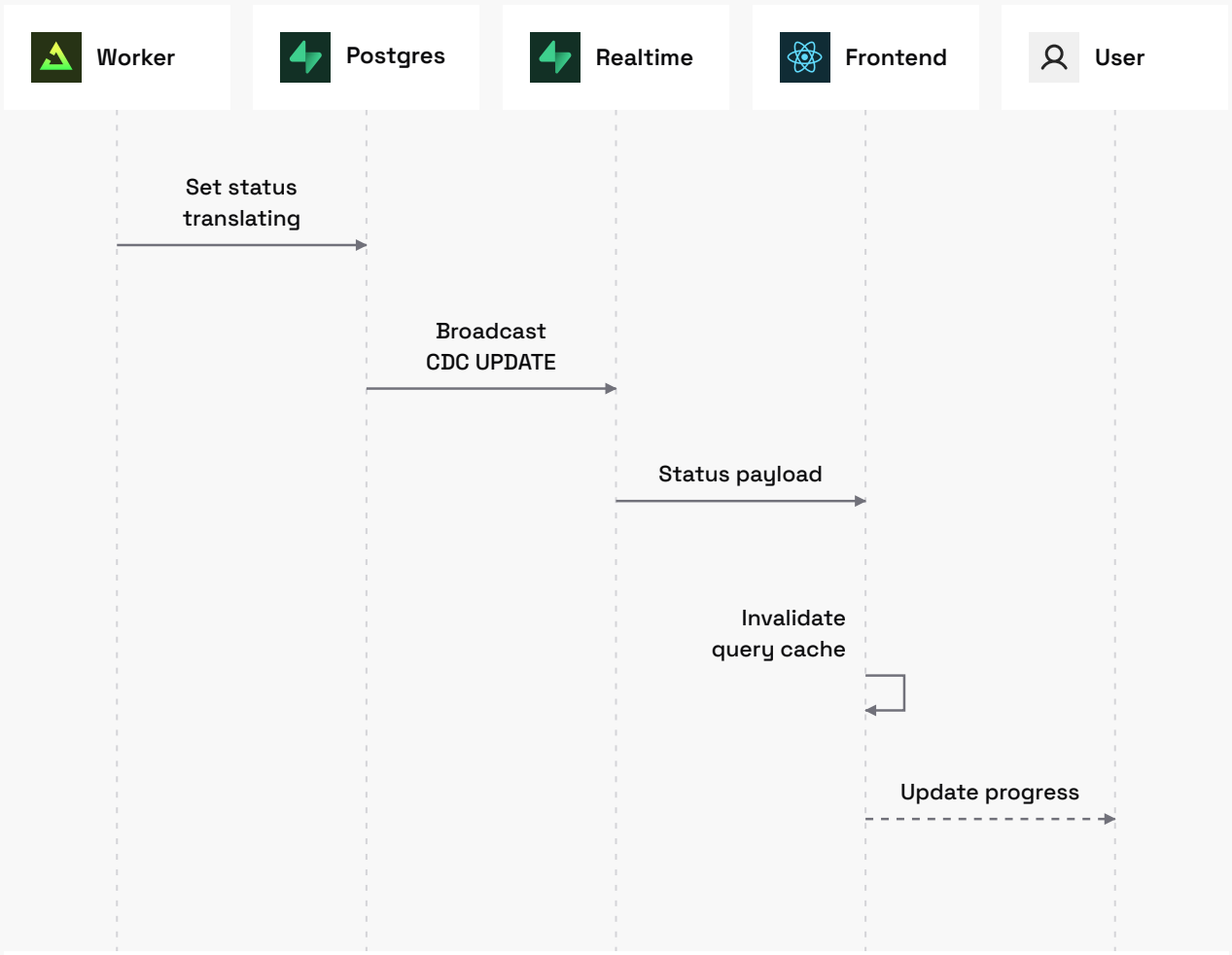
Implementation `src/hooks/use-translation-realtime.ts`

Supabase Realtime carries translation updates to the active workspace. Query refresh or polling can provide a fallback where configured.

5.1 Optimistic state and subscriptions

The shown subscription filters `translations` events by `id = translationId`. Row-level authorization must be enforced by the database; an event filter selects a record and does not itself authorize access.

11 Realtime synchronization



Payload: new.status = ”translating”.
The record filter selects a translation; RLS controls access.

5.2 Subscription hook

The hook manages the subscription channel, record filter, and cleanup.

SOURCE EXCERPT 36 · TYPESCRIPT LINES 1–18

```ts
// src/hooks/use-translation-realtime.ts
export function useTranslationRealtime(translationId: string, initialData:
TranslationRow | null) {
    const channel = supabase
        .channel(`translation:${translationId}`)
        .on(
            'postgres_changes',
            {
                event: 'UPDATE',
                filter: `id=eq.${translationId}`, // Specific Row Subscription
                schema: 'public',
                table: 'translations',
            },
            (payload) => {
                if (payload.new) setTranslation(payload.new);
            },
        )
        .subscribe();
}
```

Event delivery

Realtime events update the active translation without continuous polling. Latency depends on region, connection, and load.

Record selection

The subscription filters events to the translation being viewed.

Authorization

RLS governs which rows the connected user may access. The ID filter is not a substitute for that policy.

6. Workspace resilience

Implementation `src/components/translations/editor-skeleton.tsx`

Long document trees take time to parse and render. Loading and error states keep the workspace understandable during that work.

6.1 Loading skeleton

`EditorSkeleton` reserves space for the toolbar and document area while the editor loads, reducing visible movement during hydration.

SOURCE EXCERPT 37 · TYPESCRIPT LINES 1–15

```tsx
// src/components/translations/editor-skeleton.tsx
export function EditorSkeleton() {
    return (
        <div className="flex flex-col h-full">
             {/* Mimics Toolbar Height */}
            <div className="h-14 border-b ...">
                <Skeleton className="h-8 w-24" /> ...
            </div>
             {/* Mimics Content Padding */}
            <div className="max-w-4xl px-12 py-12 ...">
                <Skeleton className="h-4 w-full" /> ...
            </div>
        </div>
    )
}
```

6.2 Error boundaries

Route error boundaries expose recovery actions when a workspace segment fails. The documented editor fallback offers access to the original document.

7. Workspace technology

COMPONENT	TECH	ROLE
Editor Core	**Tiptap with React**	Headless content editing engine.
State	`zustand` + `react-query`	Managing editor focus modes and async data fetching.
Styling	**Tailwind Merge**	Dynamic class construction for custom node views.
Realtime	**Supabase Realtime**	Websocket subscriptions for status updates.

8. Performance strategies

The workspace uses the following strategies to reduce editor and viewer work:

Memoized configuration

`useEditorConfig` reuses expensive extension configuration for the editor instance.

Fixed table layout

The `kvp-table` class uses `table-layout: fixed` to reduce layout work.

Deferred source viewer

`next/dynamic` defers the heavy PDF viewer until it is needed.

9. Editor decisions

9.1 Why Tiptap and ProseMirror

REQUIREMENT	CHOSEN APPROACH	DESIGN CONSEQUENCE
Custom document structures	ProseMirror schema extensions	Represent signatures, description lists, and other Verto-specific nodes.
Product UI integration	Headless Tiptap editor	Own the toolbar and component styling while using the editor's document model.
Structured interchange	JSON document representation	Keep a typed document tree between processing, editing, and compilation.
Collaboration path	Yjs integration capability	A possible extension path; this comparison does not establish a shipped collaborative feature.

9.2 Why optimistic saving

The documented autosave debounce is 5000ms. Pending changes still need a visible save state and failure recovery.

Local feedback

Update the local editor immediately while the save runs.

Conflict policy

The documented last-write-wins strategy favors a simple single-user workflow. Concurrent tabs or editors can overwrite changes unless the application adds version checks or conflict handling.

Save feedback

Use the `isSaving` state for pending work, and show success only after the server acknowledges the save. Preserve the draft and expose a retry action on failure.

10. User credit consumption

Implementation `src/constants/credits.ts`

The shown credit calculation is intentionally small:

RULE	IMPLEMENTATION	RATIONALE
Simple Compute	1 Page = 1 Credit	Standard rate for all documents (PDF/Image).
Minimum Floor	`Math.max(3, pages)`	Minimum 3 credits per translation to cover fixed API/Orchestration costs.

PRICING BOUNDARY

The shown `calculateTranslationCost` function charges `Math.max(3, pages)`. The complexity multipliers in System 3 are processing-cost assumptions; the excerpts do not show them applied to the user's credit bill.

PART II · THE EXPERIENCE

Governance & operations

05

Administration, permissions, finance, and reporting.

IN THIS SECTION

01 Administrative responsibilities P. 088

02 Privilege boundaries P. 088

03 User governance P. 090

04 Financial operations P. 091

05 Operational reporting P. 091

06 Administration technology P. 092

Modules	`src/app/(protected)/admin/*` · `src/actions/admin/*`

1. Administrative responsibilities

System 5 gives administrators controlled actions for user management, credit adjustments, security recovery, and reporting. Each action needs an explicit permission check, a checked write result, and an audit record; the excerpts do not prove these properties for every action.

2. Privilege boundaries

Implementation `src/utils/supabase/server.ts` (Admin Client) & `src/utils/supabase/middleware.ts`

Ordinary actions operate under the user's RLS context. Cross-tenant administration requires explicit elevation.

2.1 Service-role access

Keep the Supabase `service_role` credential in the authorized server environment.

Authentication

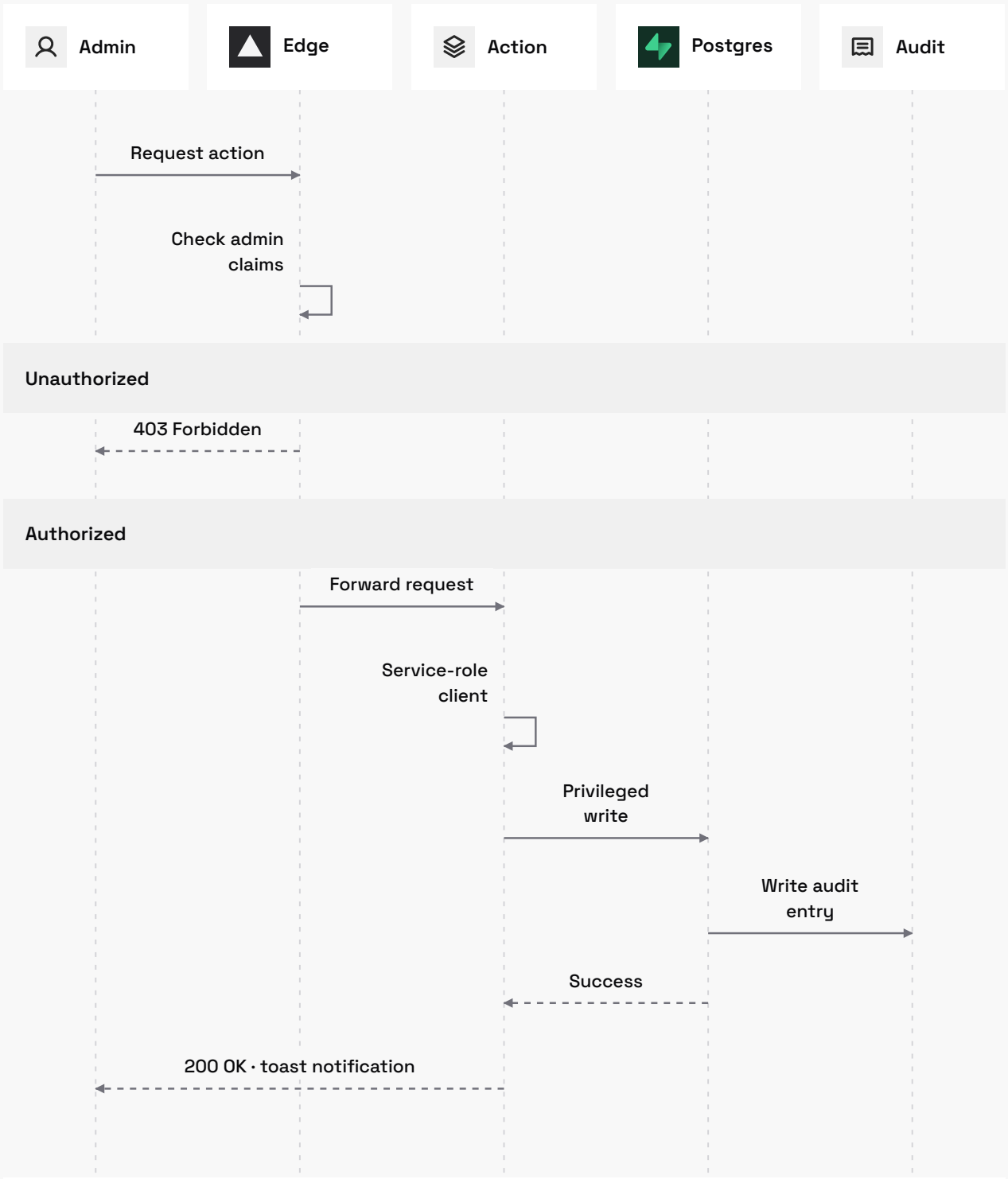
The middleware validates the `is_admin` claim or database role before permitting access to `/admin` routes.

Execution

Validated actions instantiate a `supabaseAdmin` client which bypasses RLS policies, enabling management of cross-tenant data (e.g., users, transactions, and global configurations).

2.2 Action execution flow

12 Admin action sequence



The service-role client stays server-side.
Permission, write success and audit outcome are separate checks.

3. User governance

Implementation `src/app/(protected)/admin/users`

User management provides controlled operations for account status and security recovery.

3.1 Management interface

Implementation `src/app/(protected)/admin/users/[id]/user-profile-client.tsx`

A Server Component fetches the administrative view; a Client Component handles interaction.

Query boundary

Orchestrates the "Overview" and "Billing" tabs.

Isolated state

Wraps the Billing section in a scoped `FilteringStoreProvider`, ensuring that credit transaction filters are bound to the viewed user, avoiding cross-contamination with the Admin's own session state.

3.2 Account suspension

Implementation `src/actions/admin/users/suspend-user.ts`

Suspension preserves account history while restricting access. The action and enforcement path must agree on the suspended state.

01 Update account status

Set `profiles.account_status` to `SUSPENDED` through the authorized administrative action.

02 Enforce the restriction

Middleware and protected operations must consult the account status. Verify the active-session behavior instead of assuming the status write immediately revokes all access.

03 Record the action

Write the actor, reason, and timestamp to the audit record and check the result.

3.3 MFA recovery

Implementation `src/actions/admin/users/manage-mfa.ts`

An administrator can remove a lost-device MFA factor through the provider's management API. Factor removal, session assurance, and active-session revocation are separate concerns; confirm the recovery and re-authentication policy before restoring access.

4. Financial operations

Implementation `src/actions/admin/users/adjust-credits.ts`

Financial interventions create ledger events through the normal accounting path described in System 2.

4.1 Adjustment transactions

Record each balance change as a transaction; do not directly modify `credits.amount`.

Transaction types

The shown adjustment action uses `bonus` for a credit and `penalty` for a debit. These are recognized by `sync_credits`; a generic `adjustment` type is not shown in that trigger.

Audit metadata

Include the administrative actor and reason so the adjustment can be traced.

Balance update

The normal ledger insertion path invokes `sync_credits`. Check the write result before confirming the change to the operator.

5. Operational reporting

Implementation `src/app/(protected)/admin/dashboard`

The dashboard brings processing status and credit usage together for operational review.

5.1 Processing metrics

Translation time	Tracks the duration of asynchronous jobs (System 3) to identify upstream provider delays.
Language mix	Monitors demand for specific language pairs to inform model optimization.

5.2 Credit issuance and usage

Implementation `src/actions/admin/analytics/statistics/get-credits-burn-down-analytics.ts`

Compare issued and consumed credits to understand activity and outstanding balances. Accounting treatment requires a separate policy.

Issued credits

Track credits sold or granted, distinguishing purchases from bonuses.

Consumed credits

Track credits spent on completed translation work.

Outstanding balance

Compare issued, consumed, and remaining credits. These are operational measures; the report does not determine accounting recognition.

6. Administration technology

COMPONENT	TECHNOLOGY	PURPOSE
Data Visualization	`recharts`	Rendering interactive timeseries data.
Data Fetching	**React Server Components**	Secure, server-side data retrieval.
Filter State	`zustand`	Transient state management for data tables.
Design System	**shadcn/ui**	Unified interface components.

ACCESS CONTROL Strictly limited to users with the `ADMIN` role.

PART II · THE EXPERIENCE

Design engine & brand

06

Components, communication, assets, and experience guardrails.

IN THIS SECTION

01 Product design architecture P. 094

02 Marketing and analytics P. 096

03 Product communication P. 098

04 Error and empty states P. 099

05 Launch configuration P. 099

06 Asset management P. 100

07 Component composition P. 100

08 Design and communication tools P. 101

SYSTEM RESPONSIBILITY

The product design layer defines reusable components, semantic tokens, communication patterns, and experience states. This chapter describes Verto's application UI; the handbook itself uses the Yaschet publication system.

1. Product design architecture

`src/app/globals.css` defines the product's semantic CSS tokens using Tailwind CSS v4 and the Harmony palette.

1.1 Component layers

Higher-level components consume shared tokens and primitives. This layer structure gives product teams a common starting point for color, spacing, and interaction.

Foundation

Define the base design values.

- **Tailwind configuration:** `tailwind.config.ts`.
- **Harmony palette:** OKLCH values exposed as CSS variables.

Semantic tokens

Name colors by their function.

- Use `--surface-1` and `--text-primary` instead of raw color values.
- Scope tokens to adapt the interface to light and dark themes.

Primitives

Separate behavior from appearance.

- **Radix UI:** Accessible interaction primitives.
- **CVA:** Component variants, such as `button({ intent: 'primary' })`.

Interface

Compose the foundation, tokens, and primitives into application components.

1.2 Semantic color system

OKLCH represents perceptual lightness, chroma, and hue. Verto's semantic layer maps palette values to roles such as information, success, warning, and error.

Palette foundation

The palette derives from `evilmartians/harmony`. Shared scale positions provide a useful starting relationship across hues; contrast and perceived balance still need checking in each actual component.

Surfaces

Use neutral surfaces for the canvas and structural elements.

Primary ink

Use high-contrast primary text for the core reading hierarchy.

Accent

Reserve the action color for interactive states and primary calls to action.

Feedback

Keep state colors tied to meaning.

- **Success / emerald:** Completed operations.
- **Warning / orange:** Non-blocking alerts.
- **Destructive / red:** Irreversible actions.

Typography

Sans serif	`Plus Jakarta Sans` - Modern, geometric, high-legibility.
Monospace	`Geist Mono` - Professional, minimalist.
Arabic	`Cairo` - Selected for harmony with geometric sans-serifs.

1.3 Motion

Shared motion settings define durations and easing for entry, exit, and continuous effects.

PRIMITIVE	TIMING	CURVE	USAGE
Animate In	`0.15s`	`ease-out`	Modals opening, Toasts appearing, Page transitions.
Animate Out	`0.15s`	`ease-in`	Modals closing, Items deleted.
Marquee	`var(--duration)`	`linear`	Infinite scrolling logo walls or testimonials.
Shimmer	`2s`	`linear`	Skeleton loading states for perceived performance.

1.4 Audio feedback

Short audio cues accompany selected interactions. The user's sound preference controls playback.

Sound assets	`public/sounds/`
Playback events	Success, Error, Click, Hover.
Sound library	28 custom sound effect files allowing for a rich, multi-sensory experience.

2. Marketing and analytics

The marketing layer configures search metadata, discovery, and optional analytics integrations.

2.1 Search metadata

`src/library/app.ts` centralizes application metadata:

Site constants	`APP_TITLE`, `APP_DESCRIPTION` defined once, used everywhere.
OpenGraph	Pre-generates 40+ social preview images to A/B test engagement strategies.
Robots	Granular control via `robots.txt` and meta tags for indexing (`index: true`, `follow: true`).

2.2 Discovery

Sitemap	Dynamic `sitemap.ts` automatically maps all static and dynamic routes.
Web manifest	`site.webmanifest` defines the PWA (Progressive Web App) behavior.
Structured data	JSON-LD injection for rich search results.

2.3 Performance measurement

`src/components/providers/root-provider.tsx` wires the analytics integrations:

Page analytics

The documented analytics integration records visitor activity.

Web performance

Track loading, layout stability, and interaction responsiveness. Confirm the metrics actually collected by the installed integration.

Environment gate

Enable the integration only when `NEXT_PUBLIC_ENABLE_VERCEL_ANALYTICS` is true.

3. Product communication

Toasts, transactional email, and authentication email share state terminology and brand conventions while serving different delivery contexts.

3.1 Toast notifications

`src/components/ui/sonner.tsx` adapts Sonner to the product's notification patterns.

Toast layout

Use compact floating cards and the product's `Surface-2` token.

State icons

Pair semantic feedback with the appropriate Phosphor mark, such as a check for success or a warning symbol for an error.

Theme support

Adapt notification surfaces and ink to the current light or dark theme.

3.2 Transactional email

Templates in `src/emails` use React Email to carry the product identity into transactional messages.

CTA color

Uses the exact `blue-500` (#3b82f6) from the web app for CTAs.

Email templates

- `TranslationCompleteEmail`: Clean notification with direct "Review" and "Download" actions.
- `TranslationFailedEmail`: Empathetic error reporting with support pathways.

3.3 Authentication email

Authentication templates such as magic links and sign-up confirmation live as HTML in `app/supabase/templates`.

Purpose

Send account and security messages through the authentication provider's templates. Test delivery and client compatibility.

Product identity

The documented templates use a Zinc-950 dark surface and Plus Jakarta Sans, matching Verto's application identity.

4. Error and empty states

Error, loading, and empty states explain what happened and give the reader a useful next action.

Error boundary

(`error.tsx`): Catches runtime crashes and offers "Retry" or "Support" actions.

Private loading

(`loading.tsx`): Uses `<PrivacyBlurWrapper>` to prevent sensitive data flashes during hydration.

Branded 404

(`not-found.tsx`): Redirects users to safety rather than a generic browser error.

5. Launch configuration

Feature flags in `src/app/page.tsx` select the launch experience.

Production

Renders `<ComingSoon />` with **Newsletter Lead Capture**.

Dev / staging

Renders `<Home />` (Full Marketing Site).

6. Asset management

Static brand assets live in `public`, grouped by purpose and variant.

- `public/logos`: 23 variants (Wordmark, Icon, Light/Dark) ensuring the right logo for every context.
- `public/illustrations`: 48 bespoke vector illustrations used in Empty States and Hero sections.

7. Component composition

The UI library in `src/components/ui` composes low-level primitives into reusable product components.

7.1 Composition pipeline

The component library combines headless behavior with shared visual styling:

Radix behavior

Handles state (`data-state="open"`), accessibility, and keyboard navigation.

CVA variants

Maps props (e.g., `size="sm"`) to Tailwind classes.

Theme tokens

Resolves semantic tokens (e.g., `bg-primary`) to the current theme's OKLCH values.

Motion

Applies motion to state changes (e.g., `layoutId` transitions).

Composed visual components build on those primitives:

- `cobe-globe.tsx`: WebGL 3D Globe.
- `retro-grid.tsx`: CSS-only 3D perspective grid.

8. Design and communication technology

SERVICE	TECHNOLOGY	PURPOSE
Styling	Tailwind CSS v4	Logic-based, compile-time styling.
Tokens	Evil Martians Harmony	OKLCH perceptual color generation.
Primitives	Radix UI	Headless accessibility & state.
Emails	React Email	TypeScript-based email templating.
Auth Emails	Raw HTML	High-compatibility templates.
Toasts	Sonner	High-performance notification stack.
Icons	Lucide + Phosphor	Consistent iconography.
Audio	Custom SFX	Multi-sensory feedback (`public/sounds`).
Physics	React Spring	Inertial physics for interactive elements.
Analytics	Vercel Analytics	First-party performance tracking.

PART III · INFRASTRUCTURE

Infrastructure & orchestration

07

Hosting, networking, security, and deployment.

IN THIS SECTION

01 Hosting, state, and compute P. 103

02 Network resilience P. 105

03 Environment and security P. 107

04 Deployment workflow P. 108

05 Infrastructure register P. 109

SYSTEM RESPONSIBILITY

Infrastructure separates application requests, persistent state, and background processing. This chapter covers their network limits, credentials, and deployment boundaries.

1. Hosting, state, and compute

Three services divide the main workload: Vercel hosts the application, Supabase stores and synchronizes state, and Trigger.dev executes background processing.

1.1 Application hosting

Role

Host the application and serve request-bound interactions.

Runtime shape

Favor short request handling; send document processing to background workers.

Responsibilities

- Host the documented Next.js App Router application.
- Run authentication middleware.
- Serve static assets through the edge network.

Invocation limits

The source gives conflicting historical Vercel limits here and in ADR-002. Do not treat either set as a deployment contract; check the configured plan and function runtime.

1.2 Persistent state

Role

Persist application data and synchronize state.

Database boundary

Use transactions and row-level security for tenant data.

Responsibilities

- **Postgres:** credits, users, and documents.
- **Storage:** source PDFs and processed assets.
- **Realtime:** translation state over WebSocket connections.

1.3 Background execution

Role

Run background translation and document-processing jobs.

Properties

Use durable execution for long-running work.

Responsibilities

- Run the AI orchestration DAG.
- Execute `MuPDF`, which requires binary access.
- Handle `openai` streams that exceed request timeouts.

Configuration

- **Maximum duration:** `86400s` (24 hours).
- **Machine:** Shared vCPU, as recorded in the source.

Orchestration

- **Progress:** Supabase Realtime sends WebSocket updates; TanStack Query polling provides a fallback.
- **Execution:** Background jobs run outside the edge function request lifecycle.

2. Network resilience

Long-running inference needs explicit timeout, retry, and checkpoint policies. Client configuration cannot override an upstream proxy or provider's connection limits.

2.1 HTTP client timeouts

Undici separates connection establishment, response-header, response-body, and idle-connection timeouts. Tune each for the expected workload and hosting path; a long idle timeout does not keep an active request alive indefinitely.

The source injects a custom `undici.Dispatcher` with explicit timeout values:

SOURCE EXCERPT 38 · TYPESCRIPT LINES 1–6

```ts
// src/utils/ai/openai-responses-client.ts
const dispatcher = new Agent({
    keepAliveTimeout: 30000, // 30s
    keepAliveMaxTimeout: 7200000, // 2 Hours
    headersTimeout: 7200000, // 2 Hours
});
```

`keepAliveMaxTimeout` caps the idle timeout accepted from server keep-alive hints. It does not force an upstream socket to stay open or prevent every `ECONNRESET`. The preserved excerpt is a configuration example, not proof of end-to-end connection durability.

2.2 Error taxonomy

Implementation `classify-error.ts`

Classify failures before choosing retry, recovery, or a terminal error:

ERROR	CLASS	ACTION	RECOVERY
`429 Too Many Requests`	Transient	RETRY	Exponential Backoff (1s, 2s, 4s, 8s)
`503 Service Unavailable`	Transient	RETRY	Jittered Backoff
`ETIMEDOUT`	Transient	RETRY	Retry with new TCP Socket
`400 Bad Request`	Permanent	FAIL	Immediate Refund + User Notification
`401 Unauthorized`	Permanent	FAIL	Security Alert

2.3 JSON repair

The parser attempts heuristic JSON repair after malformed or truncated output. Validate the result and reject missing required content.

01 Stack balancer

Detects unclosed `[{` and synthetically closes them.

02 Node scraper

If JSON is severely malformed, it regex-scrapes valid `ProseMirror` nodes to salvage partial work.

3. Environment and security

3.1 Environment validation

Implementation `env.ts`

`t3-oss/env-nextjs` validates environment values against a schema. Missing or invalid required values should fail configuration validation before the application serves requests.

Service-role credential

Keep `SUPABASE_SERVICE_ROLE_KEY` in the server environment. Validate it with the deployed provider configuration; a fixed string prefix is not a complete credential check.

Worker credential

Keep `TRIGGER_SECRET_KEY` in the worker's secret configuration. Validate the configured environment and permissions.

Application URL

Set `NEXT_PUBLIC_APP_URL` to the canonical application origin and use an explicit redirect allowlist.

3.2 Service-role isolation

Browser client

Authenticate the user and access only data permitted by RLS. Never distribute the service-role credential to the browser.

User server actions

Perform authorized operations in the user's context and check results.

Administrative actions

Use `supabaseAdmin` only after an explicit administrative authorization check; its elevated access bypasses RLS.

4. Deployment workflow

The frontend and background worker have separate deployment steps. Coordinate compatible versions and environment values across both services.

4.1 Frontend deployment

01 Push to main

Push the release commit with `git push origin main`.

02 Release pipeline

01 Vercel detects the commit.

02 Build the Next.js application with `pnpm build`.

03 Validate environment configuration in `env.ts`.

04 Promote the release to the edge network.

4.2 Worker deployment

Deploy background workers to Trigger.dev separately from the Vercel frontend.

01 Deploy command

Run `pnpm trigger:deploy`.

02 Verify deployment

01 Confirm that `trigger.config.ts` uses the intended `projectRef`.

02 Confirm that the Trigger.dev dashboard shows the intended version as live.

5. Infrastructure register

PROVIDER	PLAN AND SIZE	RESPONSIBILITY	COST MODEL
Vercel Pro	Serverless	Frontend / Metadata / API Routes	$20/mo/seat
Supabase Pro	8GB Disk	Relational Data / Vectors	$25/mo + Usage
Trigger.dev	Standard	Long-running Workflow Execution	Usage (Duration)
OpenAI	Tier 4	GPT-5.2 Inference	Token Usage
Google Cloud	Standard	Docs API / Drive API	Free Tier / Quota

REFERENCE · APPENDIX

Technical dependencies.

VERTO
REF
SERVICE REGISTER

The service register lists the documented providers, their responsibilities, and operational priority.

PROVIDER	SERVICE	RESPONSIBILITY	PRIORITY
Vercel	Next.js hosting	Frontend and auth middleware	Tier 1
Supabase	Postgres · Auth · Realtime	Application data and synchronization	Tier 1
Trigger.dev	Durable task runner	Background workflows	Tier 1
OpenAI	GPT-5.2 · GPT-5-mini	Language and vision inference	Tier 1
Stripe	Payments API	Revenue and credit ingestion	Tier 2
Google Cloud	Workspace · Docs API	Document generation and storage	Tier 2
Resend	Email API	Transactional notifications	Tier 3

Tier 1 — core processing · Tier 2 — payments and export · Tier 3 — notifications.

Verto translates document images into editable documents, preserving text, tables, and layout in a structured document tree.

The handbook documents the architecture behind that workflow: identity and credit integrity, vision processing, the editing workspace, and background orchestration. Diagrams, source excerpts, and decision records connect those system boundaries to their implementation.

AUTHOR

Yassine Chettouch

Product Engineer

PUBLICATION

January 2026

yaschet.com/case-studies/verto