ClaimGuru Complete Reconstruction Guide

Introduction

This document provides a comprehensive, step-by-step guide for recreating the ClaimGuru insurance CRM project. ClaimGuru is a powerful, AI-enhanced platform designed for public adjusters and insurance professionals to manage claims, clients, and business operations efficiently. This guide is intended to be used by a large language model (LLM) or a development team to rebuild the entire system from scratch, ensuring a faithful and functional reconstruction.

The information contained herein is a synthesis of extensive analysis of the original ClaimGuru application, including its architecture, database, user interface, and AI-powered features. By following these detailed instructions, a new, robust, and scalable version of ClaimGuru can be successfully developed and deployed.

1. Project Foundation

This section covers the initial setup of the ClaimGuru project, including the technology stack, project structure, dependencies, and essential configuration.

1.1. Technology Stack

The project is built on a modern, robust technology stack:

- **Frontend Framework**: React 18.3.1 with TypeScript
- **Build Tool**: Vite 6.0.1
- **Styling**: TailwindCSS with custom animations
- **Routing**: React Router DOM v6
- **State Management**: React Context API with custom hooks

- **UI Components**:
    - Radix UI primitives (`@radix-ui/react-*`)
    - Custom component library
    - Framer Motion for animations
- **Backend & Services**:
    - Supabase (Database, Authentication, Storage, Edge Functions)
    - OpenAI API for AI features
    - Google Maps API for address services
    - PDF.js for document processing
    - Tesseract.js for OCR

1.2. Project Structure

Create the following directory structure for the project:

```
/claimguru

|-- /src
|   |-- /components
|   |   |-- /admin

|   |   |-- /ai
|   |   |-- /analytics
|   |   |-- /auth

|   |   |-- /calendar
|   |   |-- /claims
|   |   |-- /clients

|   |   |-- /communication
|   |   |-- /crm
|   |   |-- /documents

|   |   |-- /finance
|   |   |-- /forms
|   |   |-- /integrations

|   |   |-- /layout
|   |   |-- /modals
|   |   |-- /ui

|   |   |-- /vendors
|   |   `-- /wizards
|   |-- /contexts

|   |-- /hooks
|   |-- /lib
|   |-- /pages

|   |-- /services
|   |-- /styles
|   `-- /utils

|-- /supabase
|   |-- /functions
|   `-- /migrations
|-- package.json
|-- tsconfig.json
|-- vite.config.ts
`-- tailwind.config.js
```

1.3. Dependencies

Create a `package.json` file with the following dependencies:

```json
{
  "name": "claimguru",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "serve": "vite preview"
  },
  "dependencies": {
    "@radix-ui/react-accordion": "^1.1.2",
    "@radix-ui/react-alert-dialog": "^1.0.5",
    "@radix-ui/react-aspect-ratio": "^1.0.3",
    "@radix-ui/react-avatar": "^1.0.4",
    "@radix-ui/react-checkbox": "^1.0.4",
    "@radix-ui/react-collapsible": "^1.0.3",
    "@radix-ui/react-context-menu": "^2.1.5",
    "@radix-ui/react-dialog": "^1.0.5",
    "@radix-ui/react-dropdown-menu": "^2.0.6",
    "@radix-ui/react-hover-card": "^1.0.7",
    "@radix-ui/react-label": "^2.0.2",
    "@radix-ui/react-menubar": "^1.0.4",
    "@radix-ui/react-navigation-menu": "^1.1.4",
    "@radix-ui/react-popover": "^1.0.7",
    "@radix-ui/react-progress": "^1.0.3",
    "@radix-ui/react-radio-group": "^1.1.3",
    "@radix-ui/react-scroll-area": "^1.0.5",
    "@radix-ui/react-select": "^2.0.0",
    "@radix-ui/react-separator": "^1.0.3",
    "@radix-ui/react-slider": "^1.1.2",
    "@radix-ui/react-slot": "^1.0.2",
    "@radix-ui/react-switch": "^1.0.3",
    "@radix-ui/react-tabs": "^1.0.4",
    "@radix-ui/react-toast": "^1.1.5",
    "@radix-ui/react-toggle": "^1.0.3",
    "@radix-ui/react-tooltip": "^1.0.7",
    "@supabase/supabase-js": "^2.39.0",
    "autoprefixer": "^10.4.16",
    "class-variance-authority": "^0.7.0",
    "clsx": "^2.0.0",
    "cmdk": "^0.2.0",
    "date-fns": "^2.30.0",
    "framer-motion": "^10.16.5",
    "lucide-react": "^0.292.0",
    "pdfjs-dist": "^3.11.174",
    "react": "^18.2.0",
    "react-day-picker": "^8.9.1",
    "react-dom": "^18.2.0",
    "react-hook-form": "^7.48.2",
    "react-router-dom": "^6.19.0",
    "tailwind-merge": "^2.0.0",
    "tailwindcss": "^3.3.5",
    "tailwindcss-animate": "^1.0.7",
    "tesseract.js": "^5.0.3",
    "zod": "^3.22.4"
  },
  "devDependencies": {
    "@types/node": "^20.9.1",
    "@types/react": "^18.2.37",
    "@types/react-dom": "^18.2.15",
    "@vitejs/plugin-react": "^4.2.0",
    "eslint": "^8.53.0",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.4",
    "typescript": "^5.2.2",
    "vite": "^5.0.0"
  }
}
```

After creating the `package.json` file, run `npm install` to install all the dependencies.

1.4. Configuration Files

1.4.1. TypeScript Configuration (`tsconfig.json`)

Create a `tsconfig.json` file with the following content:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "skipLibCheck": true,
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true,
    "baseUrl": ".",
    "paths": {
      "@/*": ["src/*"]
    }
  },
  "include": ["src"],
  "references": [{ "path": "./tsconfig.node.json" }]
}
```

1.4.2. Vite Configuration (`vite.config.ts`)

Create a `vite.config.ts` file with the following content:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import path from 'path';

export default defineConfig({
  plugins: [react()],
  resolve: {
    alias: {
      '@': path.resolve(__dirname, './src'),
    },
  },
});
```

1.4.3. Tailwind CSS Configuration (`tailwind.config.js`)

Create a `tailwind.config.js` file with the following content:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  darkMode: ['class'],
  content: [
    './pages/**/*.{ts,tsx}',
    './components/**/*.{ts,tsx}',
    './app/**/*.{ts,tsx}',
    './src/**/*.{ts,tsx}',
  ],
  theme: {
    container: {
      center: true,
      padding: '2rem',
      screens: {
        '2xl': '1400px',
      },
    },
    extend: {
      keyframes: {
        'accordion-down': {
          from: { height: 0 },
          to: { height: 'var(--radix-accordion-content-height)' },
        },
        'accordion-up': {
          from: { height: 'var(--radix-accordion-content-
height)' },
          to: { height: 0 },
        },
      },
      animation: {
        'accordion-down': 'accordion-down 0.2s ease-out',
        'accordion-up': 'accordion-up 0.2s ease-out',
      },
    },
  },
  plugins: [require('tailwindcss-animate')],
};
```

2. Database Infrastructure

This section details the complete setup of the Supabase database, including all table schemas, relationships, Row Level Security (RLS) policies, and edge functions. It is crucial to follow these specifications precisely to ensure data integrity and security.

2.1. Supabase Project Setup

1. **Create a new Supabase project.**
2. **Database Configuration**: Use the Supabase dashboard to configure the database.
3. **Enable Row Level Security (RLS)**: Enable RLS on all tables by default.

2.2. Table Schemas

The following SQL scripts should be executed in the Supabase SQL editor to create the necessary tables.

2.2.1. Core Tables

Organizations

```
CREATE TABLE organizations (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  name VARCHAR(255) NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

User Profiles

```
CREATE TABLE user_profiles (
  id UUID PRIMARY KEY REFERENCES auth.users(id),
  organization_id UUID REFERENCES organizations(id),
  role VARCHAR(50) NOT NULL, -- e.g., 'admin', 'adjuster', 'staff'
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Clients

```
CREATE TABLE clients (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  organization_id UUID NOT NULL REFERENCES organizations(id),
  created_by UUID REFERENCES auth.users(id),
  client_type VARCHAR(50) DEFAULT 'residential',
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  company_name VARCHAR(255),
  email VARCHAR(255),
  phone VARCHAR(50),
  address TEXT,
  city VARCHAR(255),
  state VARCHAR(50),
  zip_code VARCHAR(20),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Claims

```
CREATE TABLE claims (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  client_id UUID NOT NULL REFERENCES clients(id),
  organization_id UUID NOT NULL REFERENCES organizations(id),
  claim_number VARCHAR(255),
  status VARCHAR(50) DEFAULT 'open',
  date_of_loss DATE,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

2.2.2. Supporting Tables

Properties

```
CREATE TABLE properties (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  claim_id UUID NOT NULL REFERENCES claims(id),
  address TEXT,
  city VARCHAR(255),
  state VARCHAR(50),
  zip_code VARCHAR(20),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Policies

```
CREATE TABLE policies (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  claim_id UUID NOT NULL REFERENCES claims(id),
  policy_number VARCHAR(255),
  provider VARCHAR(255),
  effective_date DATE,
  expiration_date DATE,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Vendors

```
CREATE TABLE vendors (
    id uuid NOT NULL DEFAULT uuid_generate_v4() PRIMARY KEY,
    organization_id uuid NOT NULL REFERENCES organizations(id) ON
DELETE CASCADE,
    name character varying(255) NOT NULL,
    contact_person character varying(255),
    email character varying(255),
    phone character varying(50),
    website character varying(255),
    service_area text,
    created_at timestamp with time zone DEFAULT now() NOT NULL,
    updated_at timestamp with time zone DEFAULT now() NOT NULL,
    vendor_type character varying(100)
);
```

Activities

```
CREATE TABLE activities (
    id uuid NOT NULL DEFAULT uuid_generate_v4() PRIMARY KEY,
    organization_id uuid NOT NULL REFERENCES organizations(id) ON
DELETE CASCADE,
    user_id uuid REFERENCES auth.users(id) ON DELETE SET NULL,
    claim_id uuid REFERENCES claims(id) ON DELETE CASCADE,
    client_id uuid REFERENCES clients(id) ON DELETE CASCADE,
    activity_type character varying(100) NOT NULL,
    content text,
    created_at timestamp with time zone DEFAULT now() NOT NULL,
    updated_at timestamp with time zone DEFAULT now() NOT NULL
);
```

2.3. Row Level Security (RLS) Policies

RLS policies are critical for ensuring data privacy and security in a multi-tenant application. The following policies should be applied to the core tables.

Enable RLS on all tables:

```
ALTER TABLE organizations ENABLE ROW LEVEL SECURITY;
ALTER TABLE user_profiles ENABLE ROW LEVEL SECURITY;
ALTER TABLE clients ENABLE ROW LEVEL SECURITY;
ALTER TABLE claims ENABLE ROW LEVEL SECURITY;
ALTER TABLE properties ENABLE ROW LEVEL SECURITY;
ALTER TABLE policies ENABLE ROW LEVEL SECURITY;
ALTER TABLE vendors ENABLE ROW LEVEL SECURITY;
ALTER TABLE activities ENABLE ROW LEVEL SECURITY;
```

Policies for `clients` table:

```
CREATE POLICY "Allow organization members to view clients" ON
clients FOR SELECT USING (
  EXISTS (
    SELECT 1
    FROM user_profiles
    WHERE user_profiles.organization_id = clients.organization_id
      AND user_profiles.id = auth.uid()
  )
);

CREATE POLICY "Allow organization members to insert clients" ON
clients FOR INSERT WITH CHECK (
  EXISTS (
    SELECT 1
    FROM user_profiles
    WHERE user_profiles.organization_id = clients.organization_id
      AND user_profiles.id = auth.uid()
  )
);

CREATE POLICY "Allow organization members to update clients" ON
clients FOR UPDATE USING (
  EXISTS (
    SELECT 1
    FROM user_profiles
    WHERE user_profiles.organization_id = clients.organization_id
      AND user_profiles.id = auth.uid()
  )
);
```

Policies for `claims` table:

```
CREATE POLICY "Allow organization members to view claims" ON
claims FOR SELECT USING (
  EXISTS (
    SELECT 1
    FROM user_profiles
    WHERE user_profiles.organization_id = claims.organization_id
      AND user_profiles.id = auth.uid()
  )
);

CREATE POLICY "Allow organization members to insert claims" ON
claims FOR INSERT WITH CHECK (
  EXISTS (
    SELECT 1
    FROM user_profiles
    WHERE user_profiles.organization_id = claims.organization_id
      AND user_profiles.id = auth.uid()
  )
);

CREATE POLICY "Allow organization members to update claims" ON
claims FOR UPDATE USING (
  EXISTS (
    SELECT 1
    FROM user_profiles
    WHERE user_profiles.organization_id = claims.organization_id
      AND user_profiles.id = auth.uid()
  )
);
```

Note: Similar policies should be created for all other tables to enforce organization-level data isolation.

2.4. Edge Functions

Supabase Edge Functions are used for handling business logic that shouldn't be exposed on the client-side. The following edge functions need to be created.

2.4.1. `create-admin-user`

This function handles the creation of an initial admin user and an organization during the signup process.

Path: `supabase/functions/create-admin-user/index.ts`

```
import { serve } from 'https://deno.land/std@0.177.0/http/
server.ts';
import { createClient } from 'https://esm.sh/@supabase/supabase-
js@2.39.0';

serve(async (req) => {
  const { email, password, orgName } = await req.json();

  const supabaseAdmin = createClient(
    Deno.env.get('SUPABASE_URL') ?? '',
    Deno.env.get('SUPABASE_SERVICE_ROLE_KEY') ?? ''
  );

  // Create organization
  const { data: org, error: orgError } = await supabaseAdmin
    .from('organizations')
    .insert({ name: orgName })
    .select()
    .single();

  if (orgError) {
    return new Response(JSON.stringify({ error:
orgError.message }), {
      status: 500,
      headers: { 'Content-Type': 'application/json' },
    });
  }

  // Create user
  const { data: user, error: userError } = await
supabaseAdmin.auth.signUp({
    email,
    password,
  });
  if (userError) {
    return new Response(JSON.stringify({ error:
userError.message }), {
      status: 500,
      headers: { 'Content-Type': 'application/json' },
    });
  }

  // Create user profile
  const { error: profileError } = await supabaseAdmin
    .from('user_profiles')
    .insert({
      id: user.user.id,
      organization_id: org.id,
      role: 'admin',
    });

  if (profileError) {
    return new Response(JSON.stringify({ error:
profileError.message }), {
      status: 500,
      headers: { 'Content-Type': 'application/json' },
    });
  }

  return new Response(JSON.stringify({ user: user.user }), {
    headers: { 'Content-Type': 'application/json' },
  });
});
```

2.4.2. `openai-extract-fields`

This function integrates with the OpenAI API to extract structured data from unstructured text, such as that from a PDF document.

Path: `supabase/functions/openai-extract-fields/index.ts`

```
import { serve } from 'https://deno.land/std@0.177.0/http/
server.ts';
import { OpenAI } from 'https://esm.sh/openai@4.20.1';

const openai = new OpenAI({
  apiKey: Deno.env.get('OPENAI_API_KEY'),
});

serve(async (req) => {
  const { text, prompt } = await req.json();

  const completion = await openai.chat.completions.create({
    model: 'gpt-3.5-turbo',
    messages: [
      { role: 'system', content: prompt },
      { role: 'user', content: text },
    ],
  });

  const extractedData =
JSON.parse(completion.choices[0].message.content);

  return new Response(JSON.stringify(extractedData), {
    headers: { 'Content-Type': 'application/json' },
  });
});
```

3. Frontend Architecture

This section outlines the frontend architecture of the ClaimGuru application. It covers the React application structure, routing configuration, state management with contexts and hooks, and the overall organization of components.

3.1. React Application Structure

The main entry point of the application is `src/main.tsx`, which renders the `App` component.

`src/main.tsx`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './index.css';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

`src/App.tsx`

The `App` component sets up the application's routing and global providers.

```
import { BrowserRouter as Router, Routes, Route, Navigate } from
'react-router-dom';
import { AuthProvider } from './contexts/AuthContext';
import { ToastProvider } from './contexts/ToastContext';
import Layout from './components/layout/Layout';
import Dashboard from './pages/Dashboard';
import Claims from './pages/Claims';
import Clients from './pages/Clients';
import AuthPage from './pages/AuthPage';
import AuthCallback from './pages/AuthCallback';
import ProtectedRoute from './components/auth/ProtectedRoute';

function App() {
  return (
    <Router>
      <AuthProvider>
        <ToastProvider>
          <Routes>
            <Route path="/auth" element={<AuthPage />} />
            <Route path="/auth/callback" element={<AuthCallback /
>} />

            <Route path="/" element={<ProtectedRoute><Layout /></
ProtectedRoute>}>
              <Route index element={<Navigate to="/dashboard" />} /
>
              <Route path="dashboard" element={<Dashboard />} />
              <Route path="claims" element={<Claims />} />
              <Route path="claims/new" element={<Claims />} />
              <Route path="clients" element={<Clients />} />
              {/* ... other routes ... */}
            </Route>

            <Route path="*" element={<Navigate to="/dashboard" /
>} />
```

```
          </Routes>
        </ToastProvider>
      </AuthProvider>
    </Router>
  );
}

export default App;
```

3.2. Routing

Routing is managed by `react-router-dom`. The application uses a combination of public and protected routes.

- **Public Routes**: `/auth` and `/auth/callback` are accessible to everyone.
- **Protected Routes**: All other routes are wrapped in a `ProtectedRoute` component that checks for an active user session.

`src/components/auth/ProtectedRoute.tsx`

```
import { Navigate } from 'react-router-dom';
import { useAuth } from '../../contexts/AuthContext';

const ProtectedRoute = ({ children }) => {
  const { user, loading } = useAuth();

  if (loading) {
    return <div>Loading...</div>; // Or a loading spinner
  }

  if (!user) {
    return <Navigate to="/auth" />;
  }

  return children;
};

export default ProtectedRoute;
```

3.3. State Management

Global state is managed using React's Context API, with custom hooks providing access to the state and dispatcher functions.

3.3.1. `AuthContext`

Manages user authentication state, including the user object and session information.

`src/contexts/AuthContext.tsx`

```
import { createContext, useContext, useState, useEffect } from
'react';
import { supabase } from '../lib/supabase';

const AuthContext = createContext(null);

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const getSession = async () => {
      const { data: { session } } = await
supabase.auth.getSession();
      setUser(session?.user ?? null);
      setLoading(false);
    };

    getSession();

    const { data: authListener } =
supabase.auth.onAuthStateChange((_event, session) => {
      setUser(session?.user ?? null);
    });

    return () => {
      authListener.subscription.unsubscribe();
    };
  }, []);

  const value = {
    user,
    loading,
    // other auth functions like signIn, signOut
  };
  return <AuthContext.Provider value={value}>{children}</
AuthContext.Provider>;
};

export const useAuth = () => useContext(AuthContext);
```

3.3.2. `ToastContext`

Manages application-wide toast notifications.

`src/contexts/ToastContext.tsx`

```
import { createContext, useContext, useState, useCallback } from
'react';

const ToastContext = createContext(null);

export const ToastProvider = ({ children }) => {
  const [toasts, setToasts] = useState([]);

  const addToast = useCallback((message, type = 'info') => {
    const id = Date.now();
    setToasts((prevToasts) => [...prevToasts, { id, message,
type }]);
    setTimeout(() => {
      setToasts((prevToasts) => prevToasts.filter((toast) =>
toast.id !== id));
    }, 5000);
  }, []);

  const value = { addToast };

  return (
    <ToastContext.Provider value={value}>
      {children}
      {/* Render Toasts Container Here */}
    </ToastContext.Provider>
  );
};

export const useToast = () => useContext(ToastContext);
```

3.4. Custom Hooks

Custom hooks encapsulate business logic and data fetching for different parts of the application.

3.4.1. `useClients`

Manages CRUD operations and state for clients.

`src/hooks/useClients.ts`

```
import { useState, useEffect, useCallback } from 'react';
import { supabase } from '../lib/supabase';
import { useAuth } from '../contexts/AuthContext';

export const useClients = () => {
  const { user } = useAuth();
  const [clients, setClients] = useState([]);
  const [loading, setLoading] = useState(false);

  const fetchClients = useCallback(async () => {
    if (!user) return;
    setLoading(true);
    const { data: userProfile } = await supabase
      .from('user_profiles')
      .select('organization_id')
      .eq('id', user.id)
      .single();

    const { data, error } = await supabase
      .from('clients')
      .select('*')
      .eq('organization_id', userProfile.organization_id)
      .order('created_at', { ascending: false });

    if (error) {
      console.error('Error fetching clients:', error);
    } else {
      setClients(data);
    }
    setLoading(false);
  }, [user]);

  useEffect(() => {
    fetchClients();
  }, [fetchClients]);
  // ... other functions for creating, updating, and deleting
clients

  return { clients, loading, fetchClients /*, ... other functions
*/ };
};
```

3.4.2. `useClaims`

Manages CRUD operations and state for claims.

`src/hooks/useClaims.ts`

```
import { useState, useEffect, useCallback } from 'react';
import { supabase } from '../lib/supabase';
import { useAuth } from '../contexts/AuthContext';

export const useClaims = () => {
  const { user } = useAuth();
  const [claims, setClaims] = useState([]);
  const [loading, setLoading] = useState(false);

  const fetchClaims = useCallback(async () => {
    if (!user) return;
    setLoading(true);
    const { data: userProfile } = await supabase
      .from('user_profiles')
      .select('organization_id')
      .eq('id', user.id)
      .single();

    const { data, error } = await supabase
      .from('claims')
      .select('*, clients(*)')
      .eq('organization_id', userProfile.organization_id)
      .order('created_at', { ascending: false });

    if (error) {
      console.error('Error fetching claims:', error);
    } else {
      setClaims(data);
    }
    setLoading(false);
  }, [user]);

  useEffect(() => {
    fetchClaims();
  }, [fetchClaims]);
  // ... other functions for creating, updating, and deleting
claims

  return { claims, loading, fetchClaims /*, ... other functions
*/ };
};
```

3.5. Component Organization

The `src/components` directory is organized by feature or domain. This modular approach keeps the codebase clean and maintainable.

- **`ui/`**: Contains base UI components like `Button`, `Input`, `Card`, etc. These are generic and can be used anywhere in the application.
- **`layout/`**: Contains layout components like `Header`, `Sidebar`, and the main `Layout` component.
- **Feature-specific directories (e.g., `claims/`, `clients/`)**: Contain components that are specific to a particular feature. For example, `claims/` might contain `ClaimDetailView`, `ClaimList`, and so on.

4. UI Component System

This section provides detailed specifications for the reusable UI components that form the foundation of the ClaimGuru user interface. These components are located in the `src/components/ui` directory and are built using a combination of Radix UI primitives and Tailwind CSS for styling.

4.1. Core UI Components

4.1.1. Button

File: `src/components/ui/Button.tsx`

Description: A versatile button component with consistent styling and variants.

Props:

```
import { cva, type VariantProps } from 'class-variance-authority';

const buttonVariants = cva(
  'inline-flex items-center justify-center rounded-md text-sm font-
medium transition-colors focus-visible:outline-none focus-
visible:ring-2 focus-visible:ring-ring focus-visible:ring-offset-2
disabled:opacity-50 disabled:pointer-events-none ring-offset-
background',
  {
    variants: {
      variant: {
        default: 'bg-primary text-primary-foreground hover:bg-
primary/90',
        destructive: 'bg-destructive text-destructive-foreground
hover:bg-destructive/90',
        outline: 'border border-input hover:bg-accent hover:text-
accent-foreground',
        secondary: 'bg-secondary text-secondary-foreground
hover:bg-secondary/80',
        ghost: 'hover:bg-accent hover:text-accent-foreground',
        link: 'underline-offset-4 hover:underline text-primary',
      },
      size: {
        default: 'h-10 py-2 px-4',
        sm: 'h-9 px-3 rounded-md',
        lg: 'h-11 px-8 rounded-md',
      },
    },
    defaultVariants: {
      variant: 'default',
      size: 'default',
    },
  }
);
```

```
export interface ButtonProps
  extends React.ButtonHTMLAttributes<HTMLButtonElement>,
    VariantProps<typeof buttonVariants> {}
```

Implementation:

```
import * as React from 'react';
import { Slot } from '@radix-ui/react-slot';
import { cva, type VariantProps } from 'class-variance-authority';
import { cn } from '@/lib/
utils'; // A utility for merging Tailwind classes

// ... buttonVariants definition ...

const Button = React.forwardRef<HTMLButtonElement, ButtonProps>(
  ({ className, variant, size, asChild = false, ...props }, ref)
=> {
    const Comp = asChild ? Slot : 'button';
    return (
      <Comp
        className={cn(buttonVariants({ variant, size,
className }))}
        ref={ref}
        {...props}
      />
    );
  }
);
Button.displayName = 'Button';

export { Button, buttonVariants };
```

4.1.2. Card

File: `src/components/ui/Card.tsx`

Description: A flexible container component for displaying content in a card format.

Props: Standard `React.HTMLAttributes<HTMLDivElement>`.

Implementation:

```
import * as React from 'react';
import { cn } from '@/lib/utils';

const Card = React.forwardRef<HTMLDivElement,
React.HTMLAttributes<HTMLDivElement>>(
  ({ className, ...props }, ref) => (
    <div
      ref={ref}

className={cn('rounded-lg border bg-card text-card-foreground
shadow-sm', className)}
      {...props}
    />
  )
);
Card.displayName = 'Card';

// Similar components for CardHeader, CardFooter, CardTitle,
CardDescription, CardContent
```

4.1.3. Input

File: `src/components/ui/Input.tsx`

Description: A styled input component for forms.

Props: Standard `React.InputHTMLAttributes<HTMLInputElement>`.

Implementation:

```
import * as React from 'react';
import { cn } from '@/lib/utils';

export interface InputProps extends
React.InputHTMLAttributes<HTMLInputElement> {}

const Input = React.forwardRef<HTMLInputElement, InputProps>(
  ({ className, type, ...props }, ref) => {
    return (
      <input
        type={type}
        className={cn(
          'flex h-10 w-full rounded-md border border-input bg-
background px-3 py-2 text-sm ring-offset-background file:border-0
file:bg-transparent file:text-sm file:font-medium placeholder:text-
muted-foreground focus-visible:outline-none focus-visible:ring-2
focus-visible:ring-ring focus-visible:ring-offset-2
disabled:cursor-not-allowed disabled:opacity-50',
          className
        )}
        ref={ref}
        {...props}
      />
    );
  }
);
Input.displayName = 'Input';

export { Input };
```

4.2. Business-Oriented UI Components

4.2.1. AddressAutocomplete

File: `src/components/ui/AddressAutocomplete.tsx`

Description: An input component that provides Google Maps-powered address autocompletion.

Props:

```
interface AddressAutocompleteProps {
  onAddressSelect: (address: any) => void;
  // ... other props for styling and configuration
}
```

Functionality:

- Integrates with the Google Maps Places API.
- Displays a dropdown of address suggestions as the user types.
- When an address is selected, it calls the `onAddressSelect` callback with the structured address components (street, city, state, zip).

4.2.2. DocumentUpload

File: `src/components/ui/DocumentUpload.tsx`

Description: A component for uploading files with drag-and-drop support.

Props:

```
interface DocumentUploadProps {
  onFilesSelect: (files: File[]) => void;
  // ... other props for styling and file type restrictions
}
```

Functionality:

- Provides a visual area for dragging and dropping files.
- Also allows file selection via a file input.
- Displays a list of selected files.
- Calls the `onFilesSelect` callback with the selected `File` objects.

4.3. Layout Components

4.3.1. Sidebar

File: `src/components/layout/Sidebar.tsx`

Description: The main navigation sidebar for the application.

Functionality:

- **Collapsible Design:** Adapts between a full-width (256px) and a collapsed (64px) state.
- **Responsive:** Becomes a full-overlay menu on mobile devices.
- **Navigation Links:** Displays navigation links with icons (from `lucide-react`) and text.
- **Submenus:** Supports expandable submenus for hierarchical navigation.
- **Active State:** Highlights the currently active navigation link.

4.3.2. Header

File: `src/components/layout/Header.tsx`

Description: The top header of the application.

Functionality:

- **User Profile:** Displays the current user's name and a dropdown menu for accessing settings and logging out.
- **Global Search:** Includes a search input for global application search.
- **Notifications:** A notification icon that, when clicked, opens a notification center.

- **Quick Actions:** A dropdown menu with shortcuts to common tasks, such as creating a new claim or client.

This is not an exhaustive list of all components but provides a solid foundation for building the ClaimGuru UI. The same principles of composition, reusability, and consistent styling should be applied to all other components in the system.

5. AI-Powered Claim Intake Wizard

This section details the implementation of the AI-Powered Claim Intake Wizard, a cornerstone feature of the ClaimGuru application. The wizard streamlines the claim creation process by leveraging AI to extract information from uploaded documents.

5.1. Wizard Framework

The wizard is built upon a `UnifiedWizardFramework` that manages steps, state, and navigation.

File: `src/components/wizards/UnifiedWizardFramework.tsx`

Props:

```
interface UnifiedWizardFrameworkProps {
  steps: WizardStep[];
  onFinish: (wizardData: any) => void;
}

interface WizardStep {
  id: string;
  title: string;
  component: React.ComponentType<any>;
}
```

Functionality:

- Renders the current step based on the active step index.

- Provides "Next" and "Back" buttons for navigation.
- Manages the overall state of the wizard, collecting data from each step.
- Calls the `onFinish` callback when the final step is completed.

5.2. Wizard Steps

The AI-Powered Claim Intake Wizard consists of several steps, each represented by a React component.

5.2.1. Step 1: Policy & Declaration Upload

Component: `src/components/claims/wizard-steps/PolicyUploadStep.tsx`

Functionality:

- Uses the `DocumentUpload` component to allow the user to upload one or more policy documents (PDFs, images).
- Once files are selected, it calls a service to process the documents.

5.2.2. Step 2: AI-Powered Data Extraction

This is not a user-facing step but a background process that happens after the files are uploaded.

Service: `src/services/hybridPdfExtractionService.ts`

Functionality:

1. **PDF Text Extraction:** Uses `pdf.js` to extract text from PDF documents.
2. **OCR Fallback:** If text extraction fails or is incomplete, it uses `Tesseract.js` for Optical Character Recognition (OCR) on the document images.
3. **OpenAI Integration:** The extracted text is sent to the `openai-extract-fields` Supabase edge function with a carefully crafted prompt to extract structured data (e.g., policy number, insured name, address, etc.).
4. **Confidence Scoring:** The AI returns the extracted data along with a confidence score for each field.

5.2.3. Step 3: Client Details Verification

Component: `src/components/claims/wizard-steps/ClientDetailsStep.tsx`

Functionality:

- Displays a form with client information fields (name, address, phone, email).
- The fields are pre-populated with the data extracted by the AI.
- Each field is wrapped in the `ConfirmedFieldWrapper` component, which shows the AI-confidence score and allows the user to confirm, edit, or reject the extracted value.
- Uses the `AddressAutocomplete` component for address fields.

5.2.4. Step 4: Insurance Information Review

Component: `src/components/claims/wizard-steps/InsuranceInfoStep.tsx`

Functionality:

- Similar to the client details step, this step displays a form with insurance information fields (policy number, provider, effective dates).
- Fields are pre-populated from the AI extraction and are user-verifiable.

5.2.5. Step 5: Claim Information

Component: `src/components/claims/wizard-steps/ClaimInfoStep.tsx`

Functionality:

- A form for entering claim-specific details, such as the date of loss and a description of the damage.
- Some fields may be pre-populated if the information was found in the uploaded documents.

5.3. AI and Service Integration

5.3.1. `hybridPdfExtractionService`

File: `src/services/hybridPdfExtractionService.ts`

Functionality:

- Orchestrates the multi-modal extraction process (pdf.js, Tesseract.js, OpenAI).
- Handles the logic of falling back to different extraction methods if one fails.
- Returns a structured JSON object with the extracted data and confidence scores.

5.3.2. `ConfirmedFieldWrapper`

File: `src/components/ui/ConfirmedFieldWrapper.tsx`

Props:

```
interface ConfirmedFieldWrapperProps {
  label: string;
  value: any;
  status: 'confirmed' | 'modified' | 'pending' | 'rejected';
  source: 'pdf' | 'ai' | 'user';
  onConfirm: () => void;
  onReject: () => void;
  children: React.ReactNode; // The input field
}
```

Functionality:

- Renders a label, the input field, and a status icon.
- Provides buttons to confirm or reject the AI-suggested value.
- The UI changes based on the confirmation status (e.g., a green checkmark for confirmed fields).

This detailed implementation of the AI-Powered Claim Intake Wizard is a key differentiator for ClaimGuru, providing a powerful and efficient workflow for public adjusters.

6. Page-by-Page Implementation

This section provides specifications for the implementation of each major page in the ClaimGuru application, including layout, data sources, and key functionality.

6.1. Dashboard (`/dashboard`)

Component: `src/pages/Dashboard.tsx`

Layout:

- Uses the main `Layout` component, which includes the `Header` and `Sidebar`.
- The main content area is a grid-based layout to display various widgets.

Data Sources:

- `useClaims` hook to get claims data for statistics.
- `useClients` hook to get client data.
- A dedicated hook or service for fetching financial summary data.
- `useActivity` hook for the recent activity feed.

Functionality:

- **Statistics Widgets:** Display key metrics such as total claims, open claims, new clients, and total settlement value. These should be visually represented with icons and large numbers.
- **Claims Status Chart:** A chart (e.g., a donut chart) showing the distribution of claims by status (open, pending, closed).
- **Recent Activity Feed:** A list of recent activities, such as new claims, status updates, and new documents.
- **Quick Action Buttons:** Buttons for common actions like "New Claim Intake," "Add New Client," and "Create Report."

6.2. Claims (`/claims`)

Component: `src/pages/Claims.tsx`

Layout:

- Main `Layout`.
- A page header with a title ("Claims") and a "New Claim Intake" button.
- A data table for listing claims.

Data Sources:

- `useClaims` hook to fetch and manage the list of claims.

Functionality:

- **Claim List Table:** Displays a list of claims with columns for claim number, client name, date of loss, status, and assigned adjuster.
- **Filtering and Sorting:** The table should support filtering by status and sorting by any column.
- **Search:** A search bar to search for claims by claim number or client name.
- **Claim Detail View:** Clicking on a claim in the table should navigate the user to a detailed view of that claim.
- **New Claim Intake:** The "New Claim Intake" button should launch the AI-Powered Claim Intake Wizard.

6.3. Clients (`/clients`)

Component: `src/pages/Clients.tsx`

Layout:

- Main `Layout`.
- A page header with a title ("Clients") and an "Add New Client" button.
- A data table for listing clients.

Data Sources:

- `useClients` hook to fetch and manage the list of clients.

Functionality:

- **Client List Table:** Displays a list of clients with columns for name, company name, email, phone, and number of claims.
- **Filtering and Sorting:** The table should support filtering and sorting.
- **Search:** A search bar to search for clients by name or company.
- **Client Detail View:** Clicking on a client should open a modal or navigate to a detailed view with the client's information and associated claims.
- **Add New Client:** The "Add New Client" button should open a modal with a form for creating a new client.

6.4. Authentication (`/auth`)

Component: `src/pages/AuthPage.tsx`

Layout:

- A centered layout without the main `Header` and `Sidebar`.
- A card containing the login and signup forms.

Data Sources:

- `useAuth` hook for authentication functions.

Functionality:

- **Login Form:** A form with email and password fields to log in an existing user.
- **Signup Form:** A form for new users to sign up, which includes fields for email, password, and organization name. The signup form should call the `create-admin-user` Supabase edge function.
- **Tabs:** Tabs to switch between the login and signup forms.

7. Services and AI Integration

This section describes the architecture of the services layer and the integration of AI-powered features throughout the ClaimGuru application. The services layer is responsible for encapsulating business logic and interacting with external APIs and the Supabase backend.

7.1. Service Layer Architecture

The services are located in the `src/services` directory and are organized by domain.

7.1.1. Core Services

- **`supabase.ts` (`src/lib/`):** Initializes and exports the Supabase client.
- **`authService.ts`:** Manages user authentication, including login, logout, and session management, by interacting with Supabase Auth.
- **`claimService.ts`:** Handles all CRUD (Create, Read, Update, Delete) operations for claims.
- **`clientService.ts`:** Handles all CRUD operations for clients.

7.1.2. Document Processing Services

- **`pdfExtractionService.ts`:** Uses `pdf.js` to extract text from PDF documents.
- **`ocrService.ts`:** Uses `Tesseract.js` to perform OCR on images.
- **`hybridPdfExtractionService.ts`:** Orchestrates the use of `pdfExtractionService` and `ocrService`, and calls the OpenAI edge function to extract structured data. This service is the core of the AI-powered document analysis.

7.1.3. AI Services

- **`claimWizardAI.ts`:** Provides AI-driven suggestions and data validation within the Claim Intake Wizard.
- **`intelligentExtractionService.ts`:** A high-level service that uses the other document processing and AI services to provide a simple interface for intelligent data extraction from any document.

7.2. AI Integration Patterns

AI is integrated into several key areas of the application to enhance user workflows.

7.2.1. AI-Powered Document Analysis

Workflow:

1. A user uploads a document (e.g., an insurance policy).
2. The `hybridPdfExtractionService` is called with the document.
3. The service extracts the text from the document (using PDF text extraction or OCR).
4. The extracted text is sent to the `openai-extract-fields` Supabase edge function.
5. The edge function calls the OpenAI API with a prompt that asks the AI to return a JSON object with specific fields (e.g., `policy_number`, `insured_name`).
6. The AI's response is parsed and returned to the client-side application.
7. The UI is populated with the extracted data, and the user is prompted to verify the information.

Example Prompt for OpenAI:

```
You are an intelligent assistant for an insurance claims
management system. Your task is to extract specific fields from
the following text, which is from an insurance policy document.
Return the data as a JSON object with the following keys:
"policy_number", "insured_name", "property_address",
"effective_date", "expiration_date". If you cannot find a value
for a field, return null for that key.

Text: [The extracted text from the document goes here]
```

7.2.2. AI-Assisted Form Filling

The `ConfirmedFieldWrapper` component is used to display AI-extracted data in forms. This provides a seamless user experience for verifying and correcting the data.

7.3. External API Integrations

- **Supabase:** Used as the primary backend for database, authentication, storage, and edge functions.
- **OpenAI:** Used for all generative AI and data extraction tasks.
- **Google Maps Places API:** Used for the `AddressAutocomplete` component to provide address suggestions and validation.

8. Security and Deployment

This final section covers the critical aspects of security and deployment for the ClaimGuru application.

8.1. Security

8.1.1. Authentication

- **Supabase Auth:** The application uses Supabase for user authentication, which provides a secure and scalable solution.
- **JWTs (JSON Web Tokens):** Supabase Auth uses JWTs to manage user sessions. These tokens are automatically handled by the Supabase client library.
- **Protected Routes:** The `ProtectedRoute` component ensures that only authenticated users can access the application's main features.

8.1.2. Authorization

- **Row Level Security (RLS):** RLS is the cornerstone of the application's authorization model. All database tables have RLS policies that restrict access to data based on the user's organization.
- **Role-Based Access Control (RBAC):** The `user_profiles` table has a `role` column (e.g., 'admin', 'adjuster'). This can be used in RLS policies to implement more granular permissions.

8.1.3. Environment Variables

- **Supabase Credentials:** The Supabase URL and anon key should be stored in environment variables and loaded into the Vite configuration.
- **API Keys:** The OpenAI API key and Google Maps API key must be stored securely. The OpenAI key should only be accessed from the Supabase edge function, not the client-side application.

8.2. Deployment

8.2.1. Frontend (Vite)

1. **Build the Application:** Run `npm run build` to create a production-ready build of the React application in the `dist` directory.
2. **Hosting:** The `dist` directory can be deployed to any static hosting service, such as Vercel, Netlify, or AWS S3.

8.2.2. Backend (Supabase)

1. **Database Migrations:** All database schema changes should be managed through migration files in the `supabase/migrations` directory.
2. **Edge Functions:** Deploy the edge functions to Supabase using the Supabase CLI: `supabase functions deploy <function-name>`.

8.2.3. Environment Configuration

- Create separate Supabase projects for development and production.

- Use different environment variables for the development and production deployments of the frontend application.