Task Implementation Guide for Frontend Developer: VerseProjection MVP

1. Overview

Role: Frontend Developer
Objective: Develop a responsive, accessible, and performant React-based web app for VerseProjection, enabling church tech volunteers to log in, configure settings, view real-time transcriptions, select from top 10 verse matches, override misdetections, and project verses. Support offline functionality via an Electron app and ensure seamless integration with the backend (Node.js/Express, WebSocket) and AI pipeline (FastAPI). Achieve <2-second end-to-end latency, WCAG 2.1 Level AA compliance, and scalability for 1,000–10,000 users.
Scope:

- **Web App**: Login Page, Admin Dashboard (settings, transcription, matches table, override search, detection history), Projection Window, Error Notifications.
- **Offline App**: Electron app mirroring web app UI, with SQLite sync.
- **Deliverables**: React codebase, Electron integration, UI component library, accessibility audit, test suites, documentation.

  Tools: React, Material-UI, Tailwind CSS, Socket.IO, Axios, Jest, Cypress, Figma, GitHub, Electron, Vite.

2. Frontend Requirements

- **Performance**:
  - Component render: <200ms.
  - WebSocket updates: <100ms (transcription, matches).
  - End-to-end latency: <2 seconds (audio to projection).
- **Accessibility**: WCAG 2.1 Level AA (4.5:1 text contrast, keyboard navigation, ARIA labels).

- **Responsiveness**: Support 1366x768 to 1920x1080 laptops, 720p/1080p projectors.
- **Compatibility**: Chrome, Firefox (latest versions).
- **Scalability**: Handle 1,000–10,000 concurrent users (WebSocket).
- **Security**: JWT authentication, HTTPS, secure WebSocket (wss://).
- **Offline**: Electron app with identical UI, SQLite sync for feedback.

3. Frontend Architecture

- **Framework**: React (18.x, Vite for build).
- **UI Library**: Material-UI (components), Tailwind CSS (styling).
- **State Management**: React Context (user settings, matches), Zustand (optional for complex state).
- **Networking**:
    - REST: Axios for APIs (`/auth`, `/settings`, `/feedback`, `/verses/search`).
    - WebSocket: Socket.IO for audio streaming, transcription, matches.
- **Testing**: Jest (unit), Cypress (end-to-end).
- **Electron**: Reuse React components, integrate with SQLite via Node.js bindings.
- **Design System**: Based on UI/UX specs (Roboto, #1976D2 primary, 8px grid).

4. Task Breakdown

The Frontend Developer's tasks are organized into five phases: Project Setup, Web App Development, Electron Integration, Accessibility, and Testing/Optimization.

4.1 Project Setup

Objective: Initialize the React project, configure dependencies, and establish integration with backend APIs and WebSocket.
Tasks:

1. **Project Initialization**:
    - Tool: Vite (React template, TypeScript).

- Structure:
  - `src/components`: Reusable components (`Button`, `MatchesTable`, `TranscriptionView`).
  - `src/pages`: Page components (`Login`, `Dashboard`, `Projection`).
  - `src/services`: API/WebSocket handlers (`authService.ts`, `socketService.ts`).
  - `src/styles`: Tailwind CSS config, global styles.
  - `src/tests`: Jest/Cypress tests.
- Dependencies: react, react-dom, react-router-dom, @mui/material, tailwindcss, axios, socket.io-client, jwt-decode, react-hook-form.
- Build: Vite (`vite.config.ts`, minify CSS/JS).

2. **Design System Integration**:
   - **Typography**: Roboto (via `@fontsource/roboto`).
     - Headings: Roboto Bold, 24–36pt.
     - Body: Roboto Regular, 14–18pt.
     - Verse: Roboto Medium, 12–48pt.
   - **Colors**:
     - Primary: #1976D2 (blue).
     - Secondary: #388E3C (green).
     - Background: #F5F5F5 (light), #121212 (dark).
     - Text: #212121 (light), #FFFFFF (dark).
     - Error: #D32F2F (red).

Tailwind Config: `tailwind.config.js`.

```
module.exports = {
 content: ["./src/**/*.{js,ts,jsx,tsx}"],
 theme: {
   extend: {
     colors: {
       primary: "#1976D2",
       secondary: "#388E3C",
       background: { light: "#F5F5F5", dark: "#121212" },
       text: { light: "#212121", dark: "#FFFFFF" },
       error: "#D32F2F",
      },
      fontFamily: { roboto: ["Roboto", "sans-serif"] },
      spacing: { 2: "8px", 4: "16px", 6: "24px" },
    },
  },
};
```

-

Material-UI Theme: `src/theme.ts`.

```
 import { createTheme } from "@mui/material";
export const theme = createTheme({
  palette: {
    primary: { main: "#1976D2" },
    secondary: { main: "#388E3C" },
    error: { main: "#D32F2F" },
    background: { default: "#F5F5F5", paper: "#FFFFFF" },
  },
  typography: {
    fontFamily: "Roboto, Arial, sans-serif",
    h1: { fontSize: "36px", fontWeight: 700 },
    body1: { fontSize: "16px" },
  },
  spacing: 8,
});
```

-

3. **API/WebSocket Setup**:

Axios: Configure base URL (`https://app.verseprojection.com/api`).

```
 import axios from "axios";
const api = axios.create({
  baseURL: process.env.REACT_APP_API_URL,
  headers: { "Content-Type": "application/json" },
});
api.interceptors.request.use((config) => {
  const token = localStorage.getItem("token");
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});
export default api;
```

    -

Socket.IO: Connect to WebSocket (`wss://app.verseprojection.com`).

```
 import { io } from "socket.io-client";
const socket = io(process.env.REACT_APP_WS_URL, {
  auth: { token: localStorage.getItem("token") },
  autoConnect: false,
});
export default socket;
```

    -

4. **Routing**:
    - Library: react-router-dom.
    - Routes:
        - `/login`: Login Page.
        - `/dashboard`: Admin Dashboard (protected).
        - `/projection`: Projection Window (protected, new browser window).

Protected routes: Check JWT (localStorage).

```
 import { Navigate, Outlet } from "react-router-dom";
import { jwtDecode } from "jwt-decode";
const ProtectedRoute = () => {
  const token = localStorage.getItem("token");
  if (!token) return <Navigate to="/login" />;
  try {
    const decoded = jwtDecode(token);
    if (decoded.exp * 1000 < Date.now()) return <Navigate to="/login" />;
    return <Outlet />;
  } catch {
    return <Navigate to="/login" />;
  }
};
```

    ◦
5. **Environment**:

`.env`:
 REACT_APP_API_URL=https://app.verseprojection.com/api
REACT_APP_WS_URL=wss://app.verseprojection.com
VITE_PUBLIC_URL=/verseprojection

    ◦
    ◦ GitHub Actions: Build/test (`workflows/build.yml`).

Deliverables:

- React project (Vite, `src/`).
- Design system (Material-UI, Tailwind, `src/theme.ts`, `tailwind.config.js`).
- API/WebSocket services (`src/services/`).
- Documentation (setup, GitHub, `docs/frontend.md`).

4.2 Web App Development

Objective: Build React components for the Login Page, Admin Dashboard, Projection Window, and Error Notifications, integrating with backend APIs and WebSocket.
Tasks:

1. **Login Page**
    - **Component**: `LoginPage.tsx`.
    - **Features**:
        - Email/password login (POST `/auth/login`).
        - SSO (Google/Microsoft, GET `/auth/sso`).
        - Dark/light mode toggle.
        - Error handling (invalid credentials).
    - **UI**:
        - Centered card (400x500px, white/#FFFFFF or dark/#121212, 16px radius).
        - Logo (100x50px, placeholder).

        - Email/Password: Material-UI TextField (full-width, 14pt Roboto).
        - SSO Buttons: Outlined (`Sign in with Google`, `Sign in with Microsoft`, 48px height).
        - Login Button: Primary (#1976D2, filled, 48px, Roboto Bold 16pt).
        - Toggle: Material-UI Switch (top-right, 32px).
        - Error: Red text (#D32F2F, 14pt, e.g., “Invalid credentials”).
    - **Interactions**:
        - Submit: Show spinner (Material-UI CircularProgress).
        - Error: Shake card (200ms CSS animation, `keyframes shake`).
        - Keyboard: Enter submits, Tab navigates.
        - SSO: Open `/auth/sso` in new tab.

Code:

```
 import { TextField, Button, Switch, CircularProgress } from "@mui/material";
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import api from "../services/api";
const LoginPage = () => {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");
  const [loading, setLoading] = useState(false);
  const [darkMode, setDarkMode] = useState(false);
  const navigate = useNavigate();
  const handleLogin = async () => {
    setLoading(true);
    try {
      const { data } = await api.post("/auth/login", { email, password });
      localStorage.setItem("token", data.token);
      navigate("/dashboard");
    } catch (err) {
      setError("Invalid credentials");
      setLoading(false);
    }
  };
  return (
    <div className={`min-h-screen ${darkMode ? "bg-dark" : "bg-light"} flex
justify-center items-center`}>
      <div className="w-[400px] p-4 bg-white dark:bg-dark rounded-lg
shadow-md">
        <Switch checked={darkMode} onChange={() => setDarkMode(!darkMode)}
/>
        <img src="/logo.png" alt="VerseProjection" className="mx-auto mb-4" />
        <TextField
          label="Email"
          fullWidth
          margin="normal"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
          inputProps={{ "aria-label": "Email input" }}
        />
        <TextField
          label="Password"
          type="password"
          fullWidth
          margin="normal"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
          inputProps={{ "aria-label": "Password input" }}
        />
        {error && <p className="text-error text-sm">{error}</p>}
        <Button
          variant="contained"
          color="primary"
          fullWidth
          disabled={loading}
          onClick={handleLogin}
          sx={{ mt: 2, height: 48 }}
        >
          {loading ? <CircularProgress size={24} /> : "Login"}
        </Button>
        <Button
          variant="outlined"
          fullWidth
          href="/auth/sso?provider=google"
          sx={{ mt: 2, height: 48 }}
        >
          Sign in with Google
        </Button>
      </div>
    </div>
  );
};
```

-
- **Accessibility**:
  - ARIA: `aria-label` on inputs, `aria-live="polite"` for errors.
  - Contrast: 4.5:1 (text), 3:1 (buttons).
  - Keyboard: Tab order (email → password → login → SSO).

2. **Admin Dashboard**
   - **Component**: `DashboardPage.tsx`.
   - **Features**:
     - Sidebar: Settings, audio input, display, confidence threshold, start/stop projection.
     - Main Panel: Transcription view, matches table, override search, detection history.
     - WebSocket: Real-time transcription, matches (every 2 seconds).
   - **UI**:
     - **Sidebar**:
       - Width: 200px, background #E0E0E0/#1E1E1E.
       - Menu: Material-UI List (icons: `Settings`, `Mic`, 16pt Roboto).
       - Start/Stop: Button (Start: #388E3C filled, Stop: #D32F2F outlined, 48px).

        - **Settings Panel**:
            - Bible Version: Material-UI Select (KJV, WEB).
            - Audio Input: Select (e.g., “USB Mic”), test button (`Mic` icon).
            - Display: Slider (font size: 12–48pt), ColorPicker (text, background), Select (font: Roboto, Arial).
            - Confidence: Slider (0.7–0.9, step 0.05).
            - Save: Button (#1976D2, 48px).
        - **Transcription View**:
            - Height: 20%, white/#FFFFFF or dark/#121212, 14pt Roboto.
            - Scrollable (5 lines, 10s history).
            - Border: 1px #E0E0E0, 8px radius.
        - **Matches Table**:
            - Columns: Reference (15%), Text (50%), Version (15%), Confidence (20%).
            - Rows: 10 max, 48px height, hover (#BBDEFB), click (green border #388E3C).
            - Material-UI DataGrid, sortable (default: confidence descending).
        - **Override Search**:
            - TextField (100%, placeholder: “John 3:16”), autocomplete dropdown.
            - Search Button: #1976D2, `Search` icon, 48px.
        - **Detection History**:
            - Timeline: Timestamp, event (e.g., “John 3:16 selected”), confidence.
            - Scrollable, 20 entries, clickable to re-project.
    - **Interactions**:
        - Start: Connect WebSocket, emit `audio_chunk` every 2 seconds.
        - Matches: Receive `matches` event, update DataGrid, click to project (open `/projection`).
        - Override: Type query, fetch suggestions (POST `/verses/search`), select to project.

- Settings: Update via PUT `/settings`, preview in Projection Window.
- Keyboard: Arrows navigate table, Enter selects, Tab cycles inputs.

Code:

```
 import { useState, useEffect } from "react";
import { DataGrid, GridColDef } from "@mui/x-data-grid";
import { Button, TextField, Autocomplete } from "@mui/material";
import socket from "../services/socket";
import api from "../services/api";
const DashboardPage = () => {
  const [transcription, setTranscription] = useState("");
  const [matches, setMatches] = useState([]);
  const [search, setSearch] = useState("");
  const [suggestions, setSuggestions] = useState([]);
  const columns: GridColDef[] = [
    { field: "reference", headerName: "Reference", width: 150 },
    { field: "text", headerName: "Text", width: 300 },
    { field: "version", headerName: "Version", width: 100 },
    { field: "confidence", headerName: "Confidence", width: 150, sortDirection:
"desc" },
  ];
  useEffect(() => {
    socket.connect();
    socket.on("matches", ({ transcription, matches }) => {
      setTranscription(transcription);
      setMatches(matches.map((m, i) => ({ id: i, ...m })));
    });
    return () => socket.disconnect();
  }, []);
  const handleSelect = async (row) => {
    await api.post("/feedback", { transcription, selected_verse_id: row.verse_id,
top_matches: matches });
    window.open(`/projection?verse_id=${row.verse_id}`, "_blank");
  };
  const handleSearch = async () => {
    const { data } = await api.post("/verses/search", { query: search });
    setSuggestions(data);
  };
  return (
    <div className="flex h-screen">
      <div className="w-[200px] bg-gray-200 p-4">
        <Button variant="contained" color="success" fullWidth>
          Start Projection
        </Button>
      </div>
      <div className="flex-1 p-4">
        <div className="h-[20%] border rounded p-4 mb-4" aria-live="polite">
          {transcription}
        </div>
        <DataGrid
          rows={matches}
          columns={columns}
          onRowClick={(params) => handleSelect(params.row)}
          initialState={{ sorting: { sortModel: [{ field: "confidence", sort: "desc" }] } }}
          sx={{ height: "50%", mb: 4 }}
        />
        <Autocomplete
          options={suggestions}
          getOptionLabel={(option) => `${option.reference}: ${option.text}`}
          renderInput={(params) => (
            <TextField {...params} placeholder="Search verse" onChange={(e) =>
setSearch(e.target.value)} />
          )}
          onChange={(_, value) => value && handleSelect(value)}
          sx={{ mb: 4 }}
        />
      </div>
    </div>
  );
};
```

○

- **Accessibility**:
    - ARIA: `aria-label="Matches table"`, `aria-live="polite"` for transcription.
    - Keyboard: Arrow keys for table, Enter for selection.
    - Contrast: 4.5:1 for text, 3:1 for buttons.

3. **Projection Window**
    - **Component**: `ProjectionPage.tsx`.
    - **Features**:
        - Display selected verse (fetched via `verse_id`).
        - Apply user settings (font size, color, background).
        - Smooth transitions (200ms fade).
    - **UI**:
        - Full-screen, centered text (Roboto Medium, 12–48pt, default #FFFFFF).
        - Reference: Roboto Bold, 80% size (e.g., 19pt).
        - Version: Roboto Regular, 50% size, bottom-right.
        - Background: Default #000000.
    - **Interactions**:
        - Fetch verse: GET `/verses?verse_id=<id>`.
        - Update: Apply settings changes (WebSocket or polling).
        - Transition: 200ms fade-in (CSS `opacity: 0 to 1`).

Code:

```
 import { useState, useEffect } from "react";
import api from "../services/api";
const ProjectionPage = () => {
  const [verse, setVerse] = useState(null);
  const urlParams = new URLSearchParams(window.location.search);
  const verseId = urlParams.get("verse_id");
  useEffect(() => {
    const fetchVerse = async () => {
      const { data } = await api.get(`/verses?verse_id=${verseId}`);
      setVerse(data);
    };
    fetchVerse();
  }, [verseId]);
  if (!verse) return null;
  return (
    <div
      className="h-screen flex flex-col justify-center items-center"
      style={{ background: verse.bg_color, color: verse.text_color }}
    >
      <p className="text-center" style={{ fontSize: verse.font_size, transition:
"opacity 0.2s" }}>
        {verse.text}
      </p>
      <p style={{ fontSize: verse.font_size * 0.8 }}>{verse.reference}</p>
      <p className="absolute bottom-4 right-4" style={{ fontSize: verse.font_size
* 0.5 }}>
        {verse.version}
      </p>
    </div>
  );
};
```

-
- **Accessibility**:
  - ARIA: `aria-live="polite"` for verse updates.
  - Contrast: Enforce 4.5:1 (validate colors).

4. **Error Notifications**
   - **Component**: `ErrorModal.tsx`.
   - **Features**:
     - Display errors (e.g., “Low audio quality”).
     - Actions: “Check Audio”, “Open Search”, “Close”.
     - Auto-dismiss after 10s.
   - **UI**:
     - Modal: 300x200px, white/#FFFFFF or dark/#121212, 16px radius.
     - Title: 16pt Roboto Bold, #D32F2F.
     - Message: 14pt Roboto, #212121/#FFFFFF.
     - Buttons: Primary (#1976D2), Close (#757575), 48px.
   - **Interactions**:
     - Show: Triggered by WebSocket `error` event or API failure.

        - Actions: Navigate to audio settings or search bar.
        - Keyboard: Esc closes, Enter triggers primary.

Code:

```
 import { Modal, Button, Typography } from "@mui/material";
import { useState } from "react";
const ErrorModal = ({ open, message, action, onClose }) => {
  return (
    <Modal open={open} onClose={onClose} aria-label={`Error: ${message}`}>
      <div className="absolute top-1/2 left-1/2 transform -translate-x-1/2
-translate-y-1/2 w-[300px] p-4 bg-white rounded-lg shadow-md">
        <Typography variant="h6" color="error">
          Error
        </Typography>
        <Typography>{message}</Typography>
        <Button variant="contained" color="primary" onClick={action} sx={{ mt: 2, mr:
2 }}>
          {action === "check" ? "Check Audio" : "Open Search"}
        </Button>
        <Button variant="outlined" onClick={onClose} sx={{ mt: 2 }}>
          Close
        </Button>
      </div>
    </Modal>
  );
};
```

    -
    - **Accessibility**:
        - ARIA: `role="alertdialog"`, `aria-label="Error: message"`.
        - Focus: Trap focus in modal (first button).
        - Contrast: 4.5:1 for text.

Deliverables:

- React components (`src/components/`, `src/pages/`).

- API/WebSocket integration (`src/services/`).
- Test suites (Jest, `tests/components/`).
- Documentation (GitHub, `docs/components.md`).

4.3 Electron Integration

Objective: Adapt the React web app for offline use in an Electron app, reusing components and syncing feedback with SQLite.
Tasks:

1. **Electron Setup**:
   - Framework: Electron (25.x).
   - Structure:
     - `electron/main.js`: Main process (window, SQLite).
     - `electron/preload.js`: Bridge for Node.js APIs.
     - Reuse `src/` from web app (React components).
   - Dependencies: electron, @electron-forge/cli, sqlite3.
   - Build: Electron Forge (Windows/macOS, ~600MB).
2. **UI Reuse**:
   - Routes: Same as web app (`/login`, `/dashboard`, `/projection`).
   - Components: Reuse `LoginPage`, `DashboardPage`, `ProjectionPage`, `ErrorModal`.
   - Modifications:
     - Replace browser APIs (e.g., `window.open`) with Electron (`BrowserWindow`).
     - Add sync status indicator (top-right, “Offline”, “Syncing”, “Synced”).

```
import { useState, useEffect } from "react";
const SyncIndicator = () => {
  const [status, setStatus] = useState("Offline");
  useEffect(() => {
    // Check connectivity, update status
    return () => {};
  }, []);
  return (
    <div className="absolute top-4 right-4 text-sm">
      {status === "Offline" && <span className="text-gray-500">Offline</span>}
      {status === "Syncing" && <span
className="text-blue-500">Syncing...</span>}
      {status === "Synced" && <span
className="text-green-500">Synced</span>}
    </div>
  );
};
```

-

3. **Offline Data**:
   - **SQLite**:
     - Schema: Mirror PostgreSQL (`Bible`, `Feedback`).
       - `Bible`: `id`, `version`, `book`, `chapter`, `verse`, `text`, `embedding` (~12MB).
       - `Feedback`: `id`, `timestamp`, `transcription`, `selected_verse_id`, `top_matches`.
     - Size: ~12MB (Bible), ~1MB (Feedback).
   - **AI Pipeline**:
     - Call Python inference (Whisper-tiny, DistilBERT) via `python-shell`.
     - Query SQLite for matches (explicit: `book, chapter, verse`, paraphrase: NumPy cosine similarity).
   - **Sync**:
     - Store feedback in SQLite (`Feedback`, `synced=0`).
     - On reconnect: POST `/sync/feedback` (batch, axios), update `synced=1`.
     - Retry: 3 attempts, 5s interval.
4. **Platform-Specific**:
   - Title bar: Minimize, maximize, close (Windows/macOS).
   - Menu: File (Quit), Edit (Settings), Help (Docs).
   - Storage Check: Verify 600MB free space (Electron `fs`).
5. **Testing**:
   - Test on Core i5, 8GB RAM (Windows 10).

    - Metrics:
        - Render: <200ms.
        - Sync: 100% success for 100 selections.
        - Storage: No crashes at 600MB limit.

Deliverables:

- Electron app (`electron/`, ~600MB).
- SQLite integration (`electron/db.js`).
- Sync logic (`src/services/sync.ts`).
- Test suite (Jest, `tests/electron/`).
- Documentation (GitHub, `docs/electron.md`).

4.4 Accessibility Implementation

Objective: Ensure WCAG 2.1 Level AA compliance across all components.
Tasks:

1. **Contrast**:
    - Text: 4.5:1 (e.g., #212121 on #F5F5F5).
    - UI Elements: 3:1 (e.g., #1976D2 buttons).
    - Tool: WebAIM Contrast Checker, Tailwind plugin (`tailwindcss-contrast`).
2. **Keyboard Navigation**:
    - Tab order: Logical (e.g., email → password → login).
    - Arrows: Navigate Matches Table, autocomplete.
    - Enter: Submit forms, select matches.
    - Esc: Close modals.
    - Test: Manual (no mouse).
3. **Screen Readers**:
    - **ARIA**:
        - Login: `aria-label="Email input"`.
        - Dashboard: `aria-label="Matches table"`, `aria-live="polite"` for transcription.
        - Projection: `aria-live="polite"` for verses.
        - Modal: `role="alertdialog"`.
    - Test: NVDA (Windows), VoiceOver (macOS).

4. **Focus Management**:
    - Trap focus in modals (Material-UI Modal `disableAutoFocus`).
    - Highlight: 2px blue outline (#1976D2).
    - Test: Ensure visible focus.
5. **Text Resizing**: Support 200% browser zoom (responsive layouts).
6. **Audit**:
    - Tools: WAVE, axe DevTools.
    - Run before handoff (Month 7), fix issues (e.g., missing ARIA).
    - Document compliance (e.g., "4.5:1 contrast verified").

Deliverables:

- Accessibility fixes (`src/components/`).
- Audit report (GitHub, `docs/accessibility.md`).
- Screen reader test results (`tests/accessibility/`).

4.5 Testing and Optimization

Objective: Validate frontend performance, accessibility, and integration, optimizing for <200ms renders and <2-second latency.
Tasks:

1. **Unit Testing**:
    - Components: Test `LoginPage`, `DashboardPage`, `ProjectionPage`, `ErrorModal` (Jest, React Testing Library).
    - Services: Test API (`axios`), WebSocket (`socket.io-client`).
    - Coverage: 90%+.
2. **End-to-End Testing**:
    - Scenarios:
        - Login → Configure settings → Start projection → Select match → Override search → Handle error.
        - Offline: Store feedback, sync on reconnect.
    - Tool: Cypress (`cypress/e2e/`).
    - Metrics:
        - Render: <200ms.
        - WebSocket: <100ms updates.
        - End-to-end: <2 seconds.

3. **Performance Optimization**:
    - **Lazy Loading**: Use `React.lazy` for `ProjectionPage`.
    - **Memoization**: Use `React.memo`, `useMemo` for Matches Table.
    - **Bundle Size**: Minimize with Vite (target: <500KB).
    - **Profiling**: Use React DevTools Profiler, optimize slow renders (e.g., DataGrid).
4. **Browser Compatibility**:
    - Test on Chrome, Firefox (latest).
    - Fix issues (e.g., WebSocket polyfills for Firefox).
5. **Pilot Testing**:
    - Deploy to 5–10 churches (Month 8).
    - Collect feedback: Usability, bugs (e.g., table navigation).
    - Metrics:
        - Success: 90%+ complete tasks (login, select match).
        - Time: <5s to select match, <30s to start projection.
        - Satisfaction: 80%+ rate "easy to use".
    - Iterate: Fix bugs (e.g., autocomplete lag), simplify UI.
6. **Documentation**:
    - Component specs (`docs/components.md`).
    - API/WebSocket usage (`docs/services.md`).
    - Electron setup (`docs/electron.md`).
    - Store in GitHub.

Deliverables:

- Test suites (Jest, Cypress, `tests/`).
- Performance benchmarks (`tests/benchmarks.md`).
- Pilot test report (`tests/pilot.md`).
- Documentation (GitHub, `docs/`).

5. Developer Collaboration

Work closely with UI/UX, backend, AI/ML, and DevOps teams:

- **UI/UX Designer**:

    - Implement Figma designs (e.g., 48px button height, 16px padding).
    - Use design system (Roboto, #1976D2).
    - Validate accessibility (WCAG 2.1).
- **Backend Developer**:
    - Consume APIs (`/auth`, `/settings`, `/feedback`, `/verses/search`).
    - Integrate WebSocket (`audio_chunk`, `matches`).
    - Support offline sync (`/sync/feedback`).
- **AI/ML Developer**:
    - Handle AI pipeline output (`{transcription, matches}`).
    - Send feedback (`selected_verse_id`).
- **DevOps Engineer**:
    - Deploy to AWS S3 (static hosting, Vite build).
    - Configure WebSocket (API Gateway).
    - Monitor performance (CloudWatch).
- **GitHub**:
    - Create issues for bugs (e.g., “Matches Table slow render”).
    - Store code (`src/`, `tests/`, `docs/`).

Deliverables: API specs, integration tests, GitHub issues, meeting notes.

6. Risk Mitigation

- **Performance**:
    - Risk: Slow renders (>200ms) impact latency.
    - Mitigation: Lazy load, memoize, profile with React DevTools.
- **Accessibility**:
    - Risk: Non-compliance reduces adoption.
    - Mitigation: Audit with WAVE/axe, test with NVDA/VoiceOver.
- **Compatibility**:
    - Risk: Chrome/Firefox differences break UI.
    - Mitigation: Test both browsers, use polyfills.
- **WebSocket**:
    - Risk: Connection failures disrupt real-time updates.
    - Mitigation: Implement reconnect logic, test with Cypress.

- **Offline**:
    - Risk: Electron app fails on low-end laptops.
    - Mitigation: Optimize bundle, test on Core i5, ensure 600MB storage.
- **User Experience**:
    - Risk: Non-technical users find UI confusing.
    - Mitigation: Follow UI/UX specs, test with 10–15 volunteers.

Deliverables: Risk assessment, mitigation plan.

7. Success Criteria

- **Performance**:
    - Render: <200ms.
    - WebSocket: <100ms updates.
    - End-to-end: <2 seconds.
- **Accessibility**: 100% WCAG 2.1 Level AA compliance.
- **Usability**: 90%+ of pilot users complete tasks without assistance.
- **Satisfaction**: 80%+ rate UI "easy to use" (5-point scale).
- **Pilot**: 5–10 churches adopt app for Sunday services.
- **Compatibility**: Zero critical bugs in Chrome/Firefox.

8. Resources

- **PRD Reference**: artifact_id: a7c7a8e1-9f68-4929-bc62-8d5d19d66186, artifact_version_id: ca07a5bd-a168-4b01-9a91-8cc217441994.
- **Tools**:
    - React (18.x), Vite (5.x), Material-UI (5.x), Tailwind CSS (3.x).
    - Axios, Socket.IO-client, react-router-dom.
    - Jest, Cypress, React Testing Library.
    - Electron (25.x), sqlite3.
    - Figma, WAVE, axe DevTools.
- **Documentation**:
    - React: https://react.dev/
    - Material-UI: https://mui.com/
    - Tailwind CSS: https://tailwindcss.com/
    - Socket.IO: https://socket.io/docs/v4/

    - Electron: https://www.electronjs.org/
- **Team Contacts**:
    - UI/UX Designer: Figma designs, accessibility.
    - Backend Developer: APIs, WebSocket.
    - AI/ML Developer: AI pipeline integration.
    - DevOps Engineer: Deployment, monitoring.
    - Product Manager: Requirements, pilot testing.

Conclusion

This guide equips the Frontend Developer to build an intuitive, accessible, and performant React-based web app for VerseProjection, supporting real-time Bible verse detection with <2-second latency and scalability for 1,000–10,000 users. The app includes a Login Page, Admin Dashboard, Projection Window, and Error Notifications, with offline support via Electron. By adhering to WCAG 2.1, integrating with backend APIs/WebSocket,