Climate Adaptation App - Requirements Document

## Executive Summary

The Climate Adaptation App is a data-driven web application designed to help users understand the situation in their current location, and compare it with elsewhere in the world. The user will obtain metrics on things like climate adaptation, number of extreme weather events per year and trends, progress towards net zero (energy mix, electric cars, support or hostility towards developing a green economy), environment (biodiversity loss, water quality, etc), happiness, economy (inflation, average wage, cost of living, inequality, etc), social wellbeing (quality of health system, quality of social welfare, levels of racism, etc), level of democracy, level of corruption, life expectancy, population growth, food security (e.g., imports vs exports). By analysing current and projected climate data, the app provides comprehensive location assessments including climate resilience scores and risk assessments based on scientific climate models.

## Project Plan and Checklist

### 1. App Description - Purpose and Features

\*\*Purpose\*\*: The Climate Adaptation App is a comprehensive location intelligence platform that aggregates and analyses data from multiple authoritative sources to help users make informed decisions about where to live, work, or invest. The app transforms complex multi-dimensional data (climate, economic, social, environmental) into clear, actionable insights through intuitive visualisations and comparative analysis tools.

\*\*Key Features\*\*:

1. \*\*Global Location Search\*\*: Autocomplete search functionality for any city or region worldwide with real-time suggestions

2. \*\*Comprehensive Analysis\*\*: 9 categories of metrics with 50+ individual indicators including climate, economy, social wellbeing, governance, and more

3. \*\*Real-time Data Integration\*\*: Live connections to 20+ authoritative data sources including UN, World Bank, IMF, and environmental agencies

4. \*\*Comparative Analysis Tools\*\*: Side-by-side comparison of up to 4 locations with visual charts and rankings

5. \*\*Personalised Weighting System\*\*: Users can adjust importance of different factors based on their priorities

6. \*\*Progressive Data Loading\*\*: Fast initial results with detailed data loading in background for optimal performance

7. \*\*Export Capabilities\*\*: Download comprehensive reports as PDF or CSV for offline use or sharing

8. \*\*Data Transparency\*\*: Clear sourcing and confidence indicators for all metrics to ensure trust

### 2. User Stories

#### 2.1 Admin User Stories

- \*\*As an admin\*\*, I want to monitor API usage across all external data sources so I can ensure we stay within rate limits

- \*\*As an admin\*\*, I want to view system health dashboards so I can quickly identify and resolve issues

- \*\*As an admin\*\*, I want to manage API keys and credentials securely so the system remains protected

- \*\*As an admin\*\*, I want to review error logs and performance metrics so I can optimise the system

- \*\*As an admin\*\*, I want to trigger manual data refreshes so I can update information when needed

- \*\*As an admin\*\*, I want to manage user feedback and bug reports so I can prioritise improvements

#### 2.2 Data Manager User Stories (Shop Manager equivalent)

- \*\*As a data manager\*\*, I want to verify data quality scores so I can ensure information accuracy

- \*\*As a data manager\*\*, I want to configure data source priorities so the system uses the best available data

- \*\*As a data manager\*\*, I want to set cache expiration times so data remains fresh but performant

- \*\*As a data manager\*\*, I want to add new data sources so we can expand our coverage

- \*\*As a data manager\*\*, I want to create data quality reports so stakeholders understand our reliability

#### 2.3 End User Stories (Customer)

- \*\*As a user\*\*, I want to search for any location globally so I can understand its current situation

- \*\*As a user\*\*, I want to see climate projections so I can understand future risks

- \*\*As a user\*\*, I want to compare multiple locations so I can make informed decisions

- \*\*As a user\*\*, I want to customise metric weights so the analysis reflects my priorities

- \*\*As a user\*\*, I want to export analysis results so I can share or reference them later

- \*\*As a user\*\*, I want to see data sources so I can trust the information provided

- \*\*As a user\*\*, I want to save favourite locations so I can track them over time

- \*\*As a user\*\*, I want to see trends over time so I can understand if locations are improving or declining

### 3. Technical Requirements

#### 3.1 Compatibility Requirements

- \*\*Browser Support\*\*:

- Chrome 90+ (released April 2021)

- Firefox 88+ (released April 2021)

- Safari 14+ (released September 2020)

- Edge 90+ (released April 2021)

- \*\*Mobile Compatibility\*\*:

- Responsive design for screens 320px to 2560px

- Touch-optimised interface elements

- iOS 14+ and Android 8+

- \*\*Backend Runtime\*\*:

- Python 3.9+ (for type hints and async features)

- Node.js 18+ (for build tools and frontend)

- \*\*Database\*\*:

- PostgreSQL 13+ with PostGIS extension

- Redis 6+ for caching

- \*\*API Standards\*\*:

- RESTful architecture

- JSON responses

- OpenAPI 3.0 specification

#### 3.2 Dependencies

\*\*Backend Dependencies (Python)\*\*:

```txt

# Core Framework

fastapi==0.104.1

uvicorn[standard]==0.24.0

pydantic==2.5.0

pydantic-settings==2.1.0

# HTTP & Async

httpx==0.25.2

aiofiles==23.2.1

asyncpg==0.29.0

# Data Processing

pandas==2.1.3

numpy==1.24.3

scikit-learn==1.3.0

# Database

sqlalchemy==2.0.23

psycopg2-binary==2.9.7

redis==5.0.1

# Security

python-jose[cryptography]==3.3.0

passlib[bcrypt]==1.7.4

python-multipart==0.0.6

# Development

pytest==7.4.0

pytest-asyncio==0.21.0

black==23.3.0

flake8==6.0.0

mypy==1.3.0

"deforestation\_rate": -0.1 }, "water": { "quality\_index": 92, "stress\_level": "low", "per\_capita\_availability": 2450 }, "air\_quality": { "pm25\_annual": 12.5, "aqi\_average": 52, "improvement\_trend": "improving" } }, "energy\_transition": { "renewable\_percentage": 42.5, "carbon\_intensity": 233, "ev\_per\_1000": 18.5, "net\_zero\_target": 2050, "policy\_support\_score": 8.5 }, "economy": { "gdp\_per\_capita": 46252, "inflation\_5yr\_avg": 2.8, "unemployment\_rate": 4.1, "cost\_of\_living\_index": 81.2, "inequality\_gini": 0.35, "economic\_stability": 8.2 }, "social\_wellbeing": { "healthcare": { "universal\_coverage": true, "beds\_per\_1000": 2.5, "life\_expectancy": 81.3, "healthcare\_index": 74.2 }, "education": { "literacy\_rate": 99.0, "pisa\_score": 504, "university\_ranking": 85 }, "safety": { "crime\_index": 44.5, "safety\_perception": 72, "homicide\_rate": 1.2 }, "social\_support": { "welfare\_coverage": 95, "pension\_adequacy": 78, "childcare\_availability": 82 } }, "governance": { "democracy\_index": 8.54, "corruption\_score": 77, "press\_freedom": 78.5, "rule\_of\_law": 85.2, "government\_effectiveness": 88.5 }, "demographics": { "population\_growth": 0.6, "median\_age": 40.5, "dependency\_ratio": 0.56, "net\_migration": 2.5, "urbanization\_rate": 84 }, "food\_security": { "import\_dependency": 48, "food\_affordability": 95, "agricultural\_sustainability": 72, "nutrition\_score": 88 }, "happiness": { "happiness\_index": 7.064, "life\_satisfaction": 7.2, "work\_life\_balance": 7.8, "social\_connections": 8.1, "mental\_health\_support": 75 }, "data\_quality": { "completeness": 0.94, "last\_updated": "2025-01-06", "confidence\_level": "high", "missing\_indicators": ["wildfire\_risk", "indigenous\_rights"] } } }

##### Location Comparison

```typescript

// Request

POST /locations/compare

{

"locations": ["London, UK", "Berlin, Germany", "Toronto, Canada"],

"metrics": ["climate", "economy", "social\_wellbeing"],

"weights": {

"climate": 0.4,

"economy": 0.3,

"social\_wellbeing": 0.3

}

}

// Response

{

"success": true,

"comparison": {

"locations": [...],

"rankings": {

"overall": {

"Toronto, Canada": 1,

"Berlin, Germany": 2,

"London, UK": 3

},

"by\_category": {...}

},

"recommendations": [

"Toronto ranks highest due to strong social systems and moderate climate risks",

"Berlin offers best energy transition progress but faces water stress",

"London has strongest governance but highest cost of living"

]

}

}

**2. External API Integrations**

**2.1 Climate & Weather APIs**

# Open-Meteo Suite (existing)

OPENMETEO\_ENDPOINTS = {

'current': 'https://api.open-meteo.com/v1/forecast',

'climate': 'https://climate-api.open-meteo.com/v1/climate',

'archive': 'https://archive-api.open-meteo.com/v1/archive',

'geocoding': 'https://geocoding-api.open-meteo.com/v1/search'

}

# Additional Climate Sources

CLIMATE\_APIS = {

'extreme\_weather': {

'url': 'https://api.reliefweb.int/v1/disasters',

'params': {'filter[country]': country\_code}

},

'sea\_level': {

'url': 'https://api.climatecentral.org/slr',

'params': {'location': coordinates}

}

}

**2.2 Environmental APIs**

ENVIRONMENT\_APIS = {

'deforestation': {

'url': 'https://api.globalforestwatch.org/v2/forest-change',

'headers': {'x-api-key': GFW\_API\_KEY}

},

'biodiversity': {

'url': 'https://api.iucnredlist.org/v3/country',

'params': {'token': IUCN\_TOKEN}

},

'water\_quality': {

'url': 'https://api.water.org/v1/quality',

'params': {'location': location\_id}

},

'air\_quality': {

'url': 'https://api.openaq.org/v2/measurements',

'params': {'city': city\_name, 'parameter': 'pm25'}

}

}

**2.3 Socioeconomic APIs**

SOCIOECONOMIC\_APIS = {

'world\_bank': {

'base\_url': 'https://api.worldbank.org/v2',

'indicators': {

'gdp\_per\_capita': 'NY.GDP.PCAP.PP.CD',

'inflation': 'FP.CPI.TOTL.ZG',

'unemployment': 'SL.UEM.TOTL.ZS',

'gini': 'SI.POV.GINI'

}

},

'transparency\_intl': {

'url': 'https://api.transparency.org/cpi',

'params': {'country': country\_code}

},

'happiness\_report': {

'url': 'https://api.worldhappiness.report/scores',

'params': {'year': current\_year}

}

}

**3. Data Processing Pipeline**

**3.1 Parallel Data Fetching**

class LocationAnalyzer:

async def analyze\_location(self, location: str) -> dict:

"""Fetch and process all location data"""

# Get coordinates first

coords = await self.geocode\_location(location)

if not coords:

raise LocationNotFoundError(location)

# Parallel fetch all data categories

tasks = {

'climate': self.fetch\_climate\_data(coords),

'environment': self.fetch\_environment\_data(coords),

'energy': self.fetch\_energy\_data(coords),

'economy': self.fetch\_economy\_data(coords),

'social': self.fetch\_social\_data(coords),

'governance': self.fetch\_governance\_data(coords),

'demographics': self.fetch\_demographics\_data(coords),

'food': self.fetch\_food\_security\_data(coords),

'happiness': self.fetch\_happiness\_data(coords)

}

results = {}

for category, task in tasks.items():

try:

results[category] = await task

except Exception as e:

logger.error(f"Failed to fetch {category}: {e}")

results[category] = self.get\_fallback\_data(category, coords)

# Calculate scores

scores = self.scoring\_engine.calculate\_scores(results)

# Compile final analysis

return {

'location': coords,

'scores': scores,

\*\*results,

'data\_quality': self.assess\_data\_quality(results)

}

**3.2 Smart Caching Strategy**

class SmartCache:

def \_\_init\_\_(self):

self.cache\_layers = {

'memory': InMemoryCache(max\_size=1000),

'redis': RedisCache(),

'database': DatabaseCache()

}

async def get\_or\_fetch(self, key: str, fetcher: callable,

ttl: int, priority: str = 'normal'):

"""Multi-layer cache with smart invalidation"""

# Check caches in order

for layer in ['memory', 'redis', 'database']:

value = await self.cache\_layers[layer].get(key)

if value:

# Promote to faster caches

await self.\_promote\_to\_faster\_caches(key, value, layer)

return value

# Fetch fresh data

value = await fetcher()

# Store in appropriate caches based on priority

if priority == 'high':

await self.\_store\_all\_layers(key, value, ttl)

elif priority == 'normal':

await self.\_store\_persistent\_layers(key, value, ttl)

else:

await self.\_store\_database\_only(key, value, ttl)

return value

**Frontend Implementation Details**

**1. Component Architecture**

**1.1 Main Dashboard Structure**

const LocationDashboard: React.FC = () => {

const [selectedLocation, setSelectedLocation] = useState<Location | null>(null);

const [analysisData, setAnalysisData] = useState<LocationAnalysis | null>(null);

const [comparisonLocations, setComparisonLocations] = useState<Location[]>([]);

const [activeView, setActiveView] = useState<'overview' | 'comparison'>('overview');

const [userWeights, setUserWeights] = useState<CategoryWeights>(defaultWeights);

return (

<div className="min-h-screen bg-gradient-to-b from-blue-50 to-green-50">

<Header>

<LocationSearch onSelect={handleLocationSelect} />

<ViewToggle active={activeView} onChange={setActiveView} />

</Header>

{activeView === 'overview' ? (

<OverviewLayout>

<LocationSummary data={analysisData} />

<MetricsGrid data={analysisData} weights={userWeights} />

<TrendsChart data={analysisData} />

</OverviewLayout>

) : (

<ComparisonLayout>

<ComparisonControls

locations={comparisonLocations}

onAdd={handleAddComparison}

onRemove={handleRemoveComparison}

/>

<RadarChart locations={comparisonLocations} />

<ComparisonTable locations={comparisonLocations} />

<WeightCustomizer weights={userWeights} onChange={setUserWeights} />

</ComparisonLayout>

)}

<DataSourcesFooter sources={analysisData?.data\_quality.sources} />

</div>

);

};

**1.2 Metric Card Component**

interface MetricCardProps {

category: MetricCategory;

data: CategoryData;

score: number;

expanded: boolean;

onToggle: () => void;

}

const MetricCard: React.FC<MetricCardProps> = ({

category,

data,

score,

expanded,

onToggle

}) => {

const icon = getCategoryIcon(category);

const color = getScoreColor(score);

const trend = calculateTrend(data);

return (

<div className={`

metric-card rounded-lg shadow-lg p-6

border-2 transition-all duration-300

${expanded ? 'col-span-2 row-span-2' : ''}

hover:shadow-xl cursor-pointer

`}

style={{ borderColor: color }}

onClick={onToggle}

>

<div className="flex justify-between items-start mb-4">

<div className="flex items-center gap-3">

<Icon icon={icon} className="w-8 h-8" style={{ color }} />

<h3 className="text-xl font-semibold">{category.label}</h3>

</div>

<div className="text-right">

<div className="text-3xl font-bold" style={{ color }}>

{score}

</div>

<TrendIndicator trend={trend} />

</div>

</div>

{!expanded ? (

<QuickStats stats={data.keyMetrics} />

) : (

<DetailedMetrics data={data} />

)}

<LastUpdated date={data.lastUpdated} quality={data.confidence} />

</div>

);

};

**1.3 Comparison Visualization**

const RadarChart: React.FC<{ locations: LocationAnalysis[] }> = ({ locations }) => {

const categories = [

'Climate', 'Environment', 'Energy', 'Economy',

'Social', 'Governance', 'Food', 'Happiness'

];

const chartData = {

labels: categories,

datasets: locations.map((loc, index) => ({

label: loc.location.name,

data: categories.map(cat => loc.scores[cat.toLowerCase()]),

borderColor: CHART\_COLORS[index],

backgroundColor: `${CHART\_COLORS[index]}20`,

pointBackgroundColor: CHART\_COLORS[index],

pointBorderColor: '#fff',

pointHoverBackgroundColor: '#fff',

pointHoverBorderColor: CHART\_COLORS[index]

}))

};

const options = {

responsive: true,

maintainAspectRatio: false,

scales: {

r: {

min: 0,

max: 100,

ticks: { stepSize: 20 },

grid: { color: 'rgba(0,0,0,0.1)' }

}

},

plugins: {

tooltip: {

callbacks: {

label: (context) => `${context.dataset.label}: ${context.parsed.r}/100`

}

}

}

};

return (

<div className="w-full h-96 p-4 bg-white rounded-lg shadow">

<Radar data={chartData} options={options} />

</div>

);

};

**2. Progressive Data Loading**

const useProgressiveDataLoad = (location: Location) => {

const [data, setData] = useState<Partial<LocationAnalysis>>({});

const [loadingStages, setLoadingStages] = useState({

critical: true,

important: true,

supplementary: true

});

useEffect(() => {

if (!location) return;

const loadData = async () => {

// Stage 1: Critical data (immediate display)

try {

const critical = await fetchCriticalData(location);

setData(prev => ({ ...prev, ...critical }));

setLoadingStages(prev => ({ ...prev, critical: false }));

} catch (error) {

console.error('Critical data failed:', error);

}

// Stage 2: Important data (1-3 seconds)

try {

const important = await fetchImportantData(location);

setData(prev => ({ ...prev, ...important }));

setLoadingStages(prev => ({ ...prev, important: false }));

} catch (error) {

console.error('Important data failed:', error);

}

// Stage 3: Supplementary data (3-5 seconds)

try {

const supplementary = await fetchSupplementaryData(location);

setData(prev => ({ ...prev, ...supplementary }));

setLoadingStages(prev => ({ ...prev, supplementary: false }));

} catch (error) {

console.error('Supplementary data failed:', error);

}

};

loadData();

}, [location]);

return { data, loadingStages };

};

**3. Error Handling & Fallbacks**

const DataFetcher = {

async fetchWithFallback<T>(

primary: () => Promise<T>,

fallback: () => Promise<T>,

errorHandler?: (error: Error) => void

): Promise<T> {

try {

return await primary();

} catch (error) {

if (errorHandler) {

errorHandler(error as Error);

}

try {

console.warn('Primary source failed, using fallback');

return await fallback();

} catch (fallbackError) {

console.error('Both primary and fallback failed');

throw new AggregateError([error, fallbackError], 'All data sources failed');

}

}

},

async fetchWithTimeout<T>(

fetcher: () => Promise<T>,

timeout: number = 5000

): Promise<T> {

const timeoutPromise = new Promise<never>((\_, reject) => {

setTimeout(() => reject(new Error('Request timeout')), timeout);

});

return Promise.race([fetcher(), timeoutPromise]);

}

};

**Testing Implementation Details**

**1. Backend Testing**

**1.1 API Integration Tests**

@pytest.mark.asyncio

class TestLocationAnalysis:

async def test\_full\_analysis\_all\_categories(self, client):

"""Test complete location analysis returns all categories"""

response = await client.post(

"/locations/analyze",

json={"location": "London, UK", "categories": ["all"]}

)

assert response.status\_code == 200

data = response.json()

# Verify all categories present

expected\_categories = [

'climate', 'environment', 'energy', 'economy',

'social\_wellbeing', 'governance', 'demographics',

'food\_security', 'happiness'

]

for category in expected\_categories:

assert category in data['data']

assert data['data']['scores'][category] is not None

assert 0 <= data['data']['scores'][category] <= 100

async def test\_comparison\_ranking(self, client):

"""Test location comparison produces correct rankings"""

response = await client.post(

"/locations/compare",

json={

"locations": ["Oslo, Norway", "Mumbai, India", "Miami, USA"],

"metrics": ["climate", "environment"],

"weights": {"climate": 0.7, "environment": 0.3}

}

)

assert response.status\_code == 200

data = response.json()

# Verify ranking structure

assert 'rankings' in data['comparison']

assert 'overall' in data['comparison']['rankings']

assert len(data['comparison']['rankings']['overall']) == 3

**1.2 Data Quality Tests**

class TestDataQuality:

def test\_missing\_data\_imputation(self):

"""Test missing data is properly imputed"""

incomplete\_data = {

'gdp\_per\_capita': 50000,

'democracy\_index': 8.5,

# Missing corruption\_index

}

imputed = DataImputation.impute\_missing(

incomplete\_data,

Location(region='Western Europe')

)

# Should estimate corruption based on democracy score

assert 'corruption\_index' in imputed

assert 70 <= imputed['corruption\_index'] <= 85

def test\_data\_validation\_ranges(self):

"""Test data validation catches out-of-range values"""

invalid\_data = {

'temperature': 150, # Invalid

'humidity': 120, # Invalid

'democracy\_index': 15 # Invalid (0-10 scale)

}

validated = DataValidator.validate(invalid\_data)

assert validated['temperature'] is None

assert validated['humidity'] is None

assert validated['democracy\_index'] is None

**2. Frontend Testing**

**2.1 Component Tests**

describe('MetricCard Component', () => {

test('displays correct score and color', () => {

const mockData = {

category: 'climate',

score: 85,

data: mockClimateData

};

render(<MetricCard {...mockData} />);

const scoreElement = screen.getByText('85');

expect(scoreElement).toBeInTheDocument();

expect(scoreElement).toHaveStyle({ color: 'rgb(34, 197, 94)' }); // green

});

test('expands to show detailed metrics', async () => {

const mockData = {

category: 'economy',

score: 72,

data: mockEconomyData,

expanded: false,

onToggle: jest.fn()

};

const { rerender } = render(<MetricCard {...mockData} />);

// Click to expand

fireEvent.click(screen.getByRole('article'));

expect(mockData.onToggle).toHaveBeenCalled();

// Rerender with expanded state

rerender(<MetricCard {...mockData} expanded={true} />);

// Should show detailed metrics

expect(screen.getByText('GDP per Capita')).toBeInTheDocument();

expect(screen.getByText('Unemployment Rate')).toBeInTheDocument();

});

});

**2.2 Integration Tests**

describe('Location Comparison Flow', () => {

test('compares multiple locations correctly', async () => {

// Mock API responses

server.use(

rest.post('/api/locations/compare', (req, res, ctx) => {

return res(ctx.json(mockComparisonResponse));

})

);

render(<App />);

// Add first location

const searchInput = screen.getByPlaceholderText('Search for a location');

await userEvent.type(searchInput, 'London');

await screen.findByText('London, United Kingdom');

fireEvent.click(screen.getByText('London, United Kingdom'));

// Switch to comparison view

fireEvent.click(screen.getByText('Compare'));

// Add second location

fireEvent.click(screen.getByText('Add Location'));

await userEvent.type(searchInput, 'Berlin');

await screen.findByText('Berlin, Germany');

fireEvent.click(screen.getByText('Berlin, Germany'));

// Verify comparison displays

await waitFor(() => {

expect(screen.getByText('Climate Resilience')).toBeInTheDocument();

expect(screen.getByText('London')).toBeInTheDocument();

expect(screen.getByText('Berlin')).toBeInTheDocument();

});

});

});

```## API Specifications

### 1. Backend Endpoints

#### 1.1 Core Endpoints

GET / # API information GET /health # Health check GET /locations/search # Location autocomplete POST /locations/analyze # Comprehensive analysis (renamed from /climate/analyze) POST /locations/compare # Compare multiple locations GET /metrics/categories # Get all available metrics GET /data-sources # List data sources and status

#### 1.2 Request/Response Formats

##### Location Search

```typescript

// Request

GET /locations/search?q={query}&limit={limit}

// Success Response (200)

{

"success": true,

"locations": [

{

"name": "London",

"country": "United Kingdom",

"admin1": "England",

"latitude": 51.5074,

"longitude": -0.1278,

"population": 8982000,

"timezone": "Europe/London",

"display\_name": "London, England, United Kingdom"

}

],

"query": "london"

}

**Comprehensive Location Analysis**

// Request

POST /locations/analyze

Content-Type: application/json

{

"location": "London, UK",

"categories": ["all"], // or specific: ["climate", "economy", "social"]

"include\_projections": true

}

// Success Response (200)

{

"success": true,

"data": {

"location": {

"name": "London",

"country": "United Kingdom",

"latitude": 51.5074,

"longitude": -0.1278,

"population": 8982000,

"timezone": "Europe/London"

},

"scores": {

"overall": 78.5,

"climate": 72.0,

"environment": 68.5,

"energy\_transition": 81.0,

"economy": 85.0,

"social\_wellbeing": 82.0,

"governance": 88.0,

"food\_security": 75.0,

"happiness": 76.0,

"demographics": 70.0

},

"climate": {

"current\_conditions": {

"temperature": 15.4,

"humidity": 76,

"precipitation\_monthly": 45.2

},

"climate\_trends": {

"temp\_increase\_since\_1990": 1.2,

"extreme\_heat\_days\_increase": 8,

"drought\_frequency\_change": 0.15

},

"extreme\_events": {

"heat\_waves\_per\_year": 2.5,

"floods\_per\_decade": 3,

"storms\_severity\_index": 6.5

},

"projections\_2050": {

"temperature\_increase": 2.8,

"sea\_level\_rise\_impact": "moderate",

"extreme\_weather\_risk": "high"

}

},

"environment": {

"biodiversity": {

"species\_threat\_level": 0.23,

"protected\_areas\_percent": 12.5,

"deforestation\_rate": -0.1## Common Issues and Solutions

### 1. CORS Issues

#### Problem

Access to fetch at 'http://localhost:8000/api' from origin 'http://localhost:3000' has been blocked by CORS policy

#### Solution

```python

# Ensure CORS middleware is properly configured

app.add\_middleware(

CORSMiddleware,

allow\_origins=[

"https://climate-migration-app.openeyemedia.net",

"http://localhost:3000",

"http://127.0.0.1:3000" # Include both localhost variants

],

allow\_credentials=True,

allow\_methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"],

allow\_headers=["\*"],

expose\_headers=["\*"]

)

**2. Redis Connection Issues**

**Problem**

redis.exceptions.ConnectionError: Error -2 connecting to redis:6379

**Solution**

# Always implement Redis fallback

def get\_redis\_client():

try:

client = redis.from\_url(settings.redis\_url)

client.ping()

return client

except Exception as e:

logger.warning(f"Redis unavailable: {e}")

return None

# Use in-memory cache as fallback

class InMemoryCache:

def \_\_init\_\_(self):

self.cache = {}

self.timestamps = {}

def get(self, key: str):

if key in self.cache:

if time.time() - self.timestamps[key] < 3600: # 1 hour TTL

return self.cache[key]

else:

del self.cache[key]

del self.timestamps[key]

return None

def setex(self, key: str, ttl: int, value: str):

self.cache[key] = value

self.timestamps[key] = time.time()

**3. API Rate Limiting**

**Problem**

HTTPStatusError: 429 Too Many Requests

**Solution**

# Implement exponential backoff with jitter

async def fetch\_with\_backoff(url: str, params: dict, max\_retries: int = 5):

base\_delay = 1 # Start with 1 second

for attempt in range(max\_retries):

try:

response = await client.get(url, params=params)

response.raise\_for\_status()

return response.json()

except httpx.HTTPStatusError as e:

if e.response.status\_code == 429:

# Add jitter to prevent thundering herd

delay = base\_delay \* (2 \*\* attempt) + random.uniform(0, 1)

logger.warning(f"Rate limited, waiting {delay:.1f}s")

await asyncio.sleep(delay)

else:

raise

raise Exception(f"Max retries ({max\_retries}) exceeded")

**4. Data Consistency Issues**

**Problem**

Different results for the same location on repeated requests

**Solution**

# Ensure consistent data processing

def normalize\_location\_name(name: str) -> str:

"""Normalize location names for consistent caching"""

# Remove extra spaces

name = ' '.join(name.split())

# Title case for consistency

name = name.title()

# Remove special characters

name = ''.join(c for c in name if c.isalnum() or c in ' ,-')

return name

# Round coordinates consistently

def round\_coordinates(lat: float, lon: float) -> tuple:

"""Round to 4 decimal places for cache consistency"""

return round(lat, 4), round(lon, 4)

**5. Memory Leaks in Production**

**Problem**

Backend memory usage grows over time

**Solution**

# Implement request context cleanup

from contextvars import ContextVar

request\_id\_var: ContextVar[str] = ContextVar('request\_id', default='')

@app.middleware("http")

async def cleanup\_middleware(request: Request, call\_next):

# Set request ID for logging

request\_id = str(uuid.uuid4())

request\_id\_var.set(request\_id)

try:

response = await call\_next(request)

return response

finally:

# Clean up any request-specific data

request\_id\_var.set('')

# Force garbage collection for large requests

if request.url.path == "/climate/analyze":

import gc

gc.collect()

**6. Frontend State Management Issues**

**Problem**

UI doesn't update properly after API calls

**Solution**

// Always create new objects for state updates

const analyzeLocation = async (location: LocationOption) => {

setIsLoading(true);

setError(null);

try {

const response = await fetch(`${API\_BASE\_URL}/climate/analyze`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ location: location.display\_name })

});

const data = await response.json();

if (!data.success) {

throw new Error(data.error || 'Failed to analyze location');

}

// Create new object to ensure React detects change

setAnalysisData({ ...data.data });

} catch (err) {

setError(err instanceof Error ? err.message : 'An unexpected error occurred');

// Reset data on error

setAnalysisData(null);

} finally {

setIsLoading(false);

}

};

**7. TypeScript Type Errors**

**Problem**

Type errors in production build but not in development

**Solution**

// Use strict type checking

// tsconfig.json

{

"compilerOptions": {

"strict": true,

"noImplicitAny": true,

"strictNullChecks": true,

"strictFunctionTypes": true,

"noImplicitThis": true,

"alwaysStrict": true

}

}

// Define all types explicitly

interface ApiResponse<T> {

success: boolean;

data?: T;

error?: string;

}

// Use type guards

function isClimateAnalysis(data: any): data is ClimateAnalysis {

return data

&& typeof data.location === 'object'

&& typeof data.resilience\_score === 'number'

&& Array.isArray(data.recommendations);

}

**8. Production Build Optimization**

**Problem**

Large bundle sizes affecting performance

**Solution**

// next.config.js

module.exports = {

output: 'standalone',

compress: true,

poweredByHeader: false,

// Optimize images

images: {

domains: ['climate-migration-app.openeyemedia.net'],

formats: ['image/avif', 'image/webp']

},

// Bundle analyzer

webpack: (config, { isServer }) => {

if (!isServer) {

config.resolve.alias = {

...config.resolve.alias,

'@': path.resolve(\_\_dirname, 'src')

};

}

return config;

},

// Environment variables

env: {

NEXT\_PUBLIC\_API\_URL: process.env.NEXT\_PUBLIC\_API\_URL

}

};

**Performance Optimization Strategies**

**1. Database Query Optimization**

# Use connection pooling

from sqlalchemy.pool import QueuePool

engine = create\_engine(

DATABASE\_URL,

poolclass=QueuePool,

pool\_size=10,

max\_overflow=20,

pool\_pre\_ping=True, # Verify connections before use

pool\_recycle=3600 # Recycle connections after 1 hour

)

**2. Caching Strategy**

# Multi-level caching

class CacheManager:

def \_\_init\_\_(self):

self.memory\_cache = {} # L1: In-memory

self.redis\_client = get\_redis\_client() # L2: Redis

async def get(self, key: str):

# Check memory cache first

if key in self.memory\_cache:

return self.memory\_cache[key]

# Check Redis

if self.redis\_client:

value = self.redis\_client.get(key)

if value:

# Populate memory cache

self.memory\_cache[key] = json.loads(value)

return self.memory\_cache[key]

return None

**3. API Response Compression**

# Enable gzip compression

from fastapi.middleware.gzip import GZipMiddleware

app.add\_middleware(GZipMiddleware, minimum\_size=1000)

**Security Best Practices**

**1. Input Validation**

from pydantic import BaseModel, validator, constr

class LocationRequest(BaseModel):

location: constr(min\_length=2, max\_length=200, strip\_whitespace=True)

@validator('location')

def validate\_location(cls, v):

# Prevent injection attacks

forbidden\_chars = ['<', '>', '{', '}', '|', '\\']

if any(char in v for char in forbidden\_chars):

raise ValueError('Invalid characters in location')

return v

**2. Rate Limiting**

from slowapi import Limiter

from slowapi.util import get\_remote\_address

limiter = Limiter(

key\_func=get\_remote\_address,

default\_limits=["100 per minute", "1000 per hour"]

)

app.state.limiter = limiter

@app.post("/climate/analyze")

@limiter.limit("10 per minute")

async def analyze\_location(request: LocationRequest):

# Implementation

pass

**3. Environment Variable Security**

# Use pydantic settings for validation

from pydantic\_settings import BaseSettings

class Settings(BaseSettings):

redis\_url: str = "redis://localhost:6379"

cors\_origins: list[str] = ["http://localhost:3000"]

log\_level: str = "INFO"

class Config:

env\_file = ".env"

case\_sensitive = False

@validator('redis\_url')

def validate\_redis\_url(cls, v):

if not v.startswith(('redis://', 'rediss://')):

raise ValueError('Invalid Redis URL')

return v

This expanded requirements document now includes:

1. **Detailed API response structures** with exact field names and types
2. **Complete error handling patterns** with retry logic and fallbacks
3. **Specific implementation code** for critical functions
4. **Common issues and their solutions** based on the existing codebase
5. **Testing patterns** for both backend and frontend
6. **Production configuration details** including Docker and monitoring
7. **Security best practices** with code examples
8. **Performance optimization strategies**

These additions should significantly reduce the back-and-forth during implementation by providing clear, specific guidance for handling edge cases and common problems. current = self.\_generate\_realistic\_current\_data( location\_name, lat, lon) elif i == 1: # Historical baseline historical = self.\_generate\_fallback\_baseline(lat, lon) elif i == 2: # Projections projections = self.\_generate\_fallback\_projections(lat, lon) elif i == 3: # Recent average recent = self.\_generate\_fallback\_recent(lat, lon)

# Process and combine results

analysis = self.\_combine\_analysis\_data(

location\_data, current, historical, projections, recent

)

# Cache the result

if self.use\_cache and analysis:

await self.\_save\_to\_cache(cache\_key, analysis, self.cache\_ttl['full\_analysis'])

return analysis

except Exception as e:

print(f"Comprehensive analysis failed: {e}")

return self.\_generate\_complete\_fallback(location\_data)

#### 3.2 Rate Limit Management

```python

class RateLimiter:

def \_\_init\_\_(self, max\_requests: int = 100, window\_seconds: int = 60):

self.max\_requests = max\_requests

self.window\_seconds = window\_seconds

self.requests = []

async def check\_rate\_limit(self) -> bool:

"""Check if request is within rate limit"""

now = time.time()

# Remove old requests outside window

self.requests = [req for req in self.requests

if req > now - self.window\_seconds]

if len(self.requests) >= self.max\_requests:

return False

self.requests.append(now)

return True

async def wait\_if\_needed(self):

"""Wait until rate limit allows request"""

if not await self.check\_rate\_limit():

oldest\_request = min(self.requests)

wait\_time = self.window\_seconds - (time.time() - oldest\_request) + 1

print(f"Rate limit reached, waiting {wait\_time}s")

await asyncio.sleep(wait\_time)

# Usage in service

rate\_limiter = RateLimiter(max\_requests=100, window\_seconds=60)

async def make\_api\_request(url: str, params: dict):

await rate\_limiter.wait\_if\_needed()

# Make request...

**4. Error Recovery Strategies**

**4.1 Graceful Degradation**

async def get\_current\_climate(self, latitude: float, longitude: float) -> dict:

"""Get current climate with multiple fallback levels"""

# Level 1: Try primary API

try:

data = await self.\_fetch\_openmeteo\_current(latitude, longitude)

if data:

return data

except Exception as e:

print(f"Primary API failed: {e}")

# Level 2: Try cache even if expired

if self.use\_cache:

expired\_data = await self.\_get\_expired\_cache(

f"current:{latitude:.2f}:{longitude:.2f}"

)

if expired\_data:

print("Using expired cache data")

return expired\_data

# Level 3: Try alternative API endpoint

try:

data = await self.\_fetch\_openmeteo\_forecast(latitude, longitude)

if data:

return data

except Exception as e:

print(f"Alternative API failed: {e}")

# Level 4: Generate realistic fallback

return self.\_generate\_realistic\_current\_data("Unknown", latitude, longitude)

**4.2 Data Validation**

def validate\_climate\_data(data: dict) -> bool:

"""Validate climate data for sanity"""

try:

# Temperature checks

temp = data.get('current\_temperature')

if not isinstance(temp, (int, float)) or temp < -50 or temp > 60:

return False

# Humidity checks

humidity = data.get('current\_humidity')

if humidity and (humidity < 0 or humidity > 100):

return False

# Precipitation checks

precip = data.get('total\_precipitation')

if precip and (precip < 0 or precip > 5000): # Max 5000mm/month

return False

return True

except Exception:

return False

**Testing Implementation Details**

**1. Backend Testing**

**1.1 Unit Test Structure**

# test\_climate\_service.py

import pytest

from unittest.mock import Mock, patch, AsyncMock

from app.services.climate\_service import ClimateDataService

@pytest.fixture

def climate\_service():

"""Create service without Redis for testing"""

with patch('redis.from\_url') as mock\_redis:

mock\_redis.side\_effect = Exception("No Redis in tests")

service = ClimateDataService()

assert not service.use\_cache

return service

@pytest.fixture

def mock\_api\_responses():

"""Mock API responses for testing"""

return {

'geocoding': {

"results": [{

"name": "London",

"latitude": 51.5074,

"longitude": -0.1278,

"country": "United Kingdom"

}]

},

'current': {

"current": {

"temperature\_2m": 15.4,

"relative\_humidity\_2m": 76

}

},

'historical': {

"daily": {

"time": ["1990-07-01", "1990-07-02"],

"temperature\_2m\_max": [22.5, 23.1],

"temperature\_2m\_min": [14.2, 14.8],

"precipitation\_sum": [0.0, 2.1]

}

}

}

@pytest.mark.asyncio

async def test\_location\_search(climate\_service, mock\_api\_responses):

"""Test location search functionality"""

with patch('httpx.AsyncClient.get') as mock\_get:

mock\_get.return\_value.json.return\_value = mock\_api\_responses['geocoding']

mock\_get.return\_value.raise\_for\_status = Mock()

result = await climate\_service.get\_location\_coordinates("London")

assert result['name'] == "London"

assert result['latitude'] == 51.5074

mock\_get.assert\_called\_once()

**1.2 Integration Test Pattern**

@pytest.mark.asyncio

async def test\_full\_analysis\_flow(climate\_service):

"""Test complete analysis flow with mocked APIs"""

with patch.multiple('httpx.AsyncClient',

get=AsyncMock(side\_effect=mock\_api\_sequence)):

result = await climate\_service.get\_comprehensive\_climate\_analysis(

"London, UK"

)

assert result is not None

assert result['location']['name'] == "London"

assert 'current\_climate' in result

assert 'projections' in result

assert result['resilience\_score'] >= 0

assert result['resilience\_score'] <= 100

**2. Frontend Testing**

**2.1 Component Testing**

// ClimateApp.test.tsx

import { render, screen, fireEvent, waitFor } from '@testing-library/react';

import { ClimateApp } from './ClimateApp';

// Mock fetch

global.fetch = jest.fn();

describe('ClimateApp', () => {

beforeEach(() => {

fetch.mockClear();

});

test('searches for locations on input', async () => {

const mockLocations = {

success: true,

locations: [{

name: 'London',

country: 'United Kingdom',

display\_name: 'London, United Kingdom'

}]

};

fetch.mockResolvedValueOnce({

json: async () => mockLocations

});

render(<ClimateApp />);

const searchInput = screen.getByPlaceholderText(/search for a city/i);

fireEvent.change(searchInput, { target: { value: 'Lond' } });

await waitFor(() => {

expect(fetch).toHaveBeenCalledWith(

expect.stringContaining('/locations/search?q=Lond')

);

});

});

test('displays loading state during analysis', async () => {

render(<ClimateApp />);

// Trigger analysis

const analyzeButton = screen.getByText(/analyze climate/i);

fireEvent.click(analyzeButton);

expect(screen.getByText(/analyzing/i)).toBeInTheDocument();

});

});

**2.2 Data Formatting Tests**

// formatters.test.ts

import { formatTemperature, getVariationScore } from './formatters';

describe('Data Formatters', () => {

describe('formatTemperature', () => {

test('formats valid temperatures correctly', () => {

expect(formatTemperature(15.456)).toBe('15.5°C');

expect(formatTemperature(0)).toBe('0°C');

expect(formatTemperature(-5.123)).toBe('-5.1°C');

});

test('handles invalid inputs', () => {

expect(formatTemperature(undefined)).toBe('--');

expect(formatTemperature(NaN)).toBe('--');

expect(formatTemperature(null as any)).toBe('--');

});

});

describe('getVariationScore', () => {

test('converts variations to percentage correctly', () => {

expect(getVariationScore(0, 5)).toBe(50); // No change = 50%

expect(getVariationScore(5, 5)).toBe(100); // Max positive = 100%

expect(getVariationScore(-5, 5)).toBe(0); // Max negative = 0%

expect(getVariationScore(2.5, 5)).toBe(75); // Half positive = 75%

});

});

});

**Deployment Configuration Details**

**1. Environment Variables**

**1.1 Backend Configuration**

# .env.production

PORT=8000

REDIS\_URL=redis://default:password@redis-server:6379

NODE\_ENV=production

CORS\_ORIGINS=https://climate-migration-app.openeyemedia.net

LOG\_LEVEL=info

# API endpoints (with defaults)

GEOCODING\_API\_URL=https://geocoding-api.open-meteo.com/v1

WEATHER\_API\_URL=https://api.open-meteo.com/v1

CLIMATE\_API\_URL=https://climate-api.open-meteo.com/v1

ARCHIVE\_API\_URL=https://archive-api.open-meteo.com/v1

# Rate limiting

MAX\_REQUESTS\_PER\_MINUTE=100

MAX\_REQUESTS\_PER\_HOUR=1000

**1.2 Frontend Configuration**

# .env.production

NEXT\_PUBLIC\_API\_URL=https://climate-migration-app.openeyemedia.net/api

NEXT\_PUBLIC\_ENVIRONMENT=production

NEXT\_PUBLIC\_SENTRY\_DSN=https://your-sentry-dsn

**2. Docker Configuration**

**2.1 Backend Dockerfile**

FROM python:3.9-slim

WORKDIR /app

# Install system dependencies

RUN apt-get update && apt-get install -y \

gcc \

&& rm -rf /var/lib/apt/lists/\*

# Install Python dependencies

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

# Copy application code

COPY . .

# Health check

HEALTHCHECK --interval=30s --timeout=3s --start-period=40s --retries=3 \

CMD python -c "import requests; requests.get('http://localhost:8000/health')"

# Run with uvicorn

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

**2.2 Frontend Dockerfile**

FROM node:18-alpine AS builder

WORKDIR /app

# Copy package files

COPY package\*.json ./

RUN npm ci

# Copy source code

COPY . .

# Build

RUN npm run build

# Production stage

FROM node:18-alpine

WORKDIR /app

# Copy built application

COPY --from=builder /app/.next ./.next

COPY --from=builder /app/node\_modules ./node\_modules

COPY --from=builder /app/package.json ./package.json

COPY --from=builder /app/public ./public

# Health check

HEALTHCHECK --interval=30s --timeout=3s --start-period=40s --retries=3 \

CMD node -e "require('http').get('http://localhost:3000/api/health')"

EXPOSE 3000

CMD ["npm", "start"]

**3. Production Monitoring**

**3.1 Logging Configuration**

# app/core/logging.py

import logging

import json

from datetime import datetime

class JSONFormatter(logging.Formatter):

def format(self, record):

log\_data = {

'timestamp': datetime.utcnow().isoformat(),

'level': record.levelname,

'message': record.getMessage(),

'module': record.module,

'function': record.funcName,

'line': record.lineno

}

if hasattr(record, 'request\_id'):

log\_data['request\_id'] = record.request\_id

if record.exc\_info:

log\_data['exception'] = self.formatException(record.exc\_info)

return json.dumps(log\_data)

# Configure logging

logger = logging.getLogger(\_\_name\_\_)

handler = logging.StreamHandler()

handler.setFormatter(JSONFormatter())

logger.addHandler(handler)

logger.setLevel(logging.INFO)

**3.2 Health Check Endpoints**

# app/api/health.py

from fastapi import APIRouter

from datetime import datetime

import httpx

import redis

router = APIRouter()

@router.get("/health")

async def health\_check():

"""Comprehensive health check"""

health\_status = {

"status": "healthy",

"timestamp": datetime.utcnow().isoformat(),

"checks": {}

}

# Check Redis

try:

r = redis.from\_url(settings.redis\_url)

r.ping()

health\_status["checks"]["redis"] = "healthy"

except Exception as e:

health\_status["checks"]["redis"] = f"unhealthy: {str(e)}"

health\_status["status"] = "degraded"

# Check external APIs

try:

async with httpx.AsyncClient() as client:

response = await client.get(

"https://api.open-meteo.com/v1/forecast?latitude=0&longitude=0",

timeout=5.0

)

response.raise\_for\_status()

health\_status["checks"]["openmeteo"] = "healthy"

except Exception as e:

health\_status["checks"]["openmeteo"] = f"unhealthy: {str(e)}"

health\_status["status"] = "degraded"

return health\_status

@router.get("/health/live")

async def liveness\_check():

"""Simple liveness check for k8s"""

return {"status": "alive"}

@router.get("/health/ready")

async def readiness\_check():

"""Readiness check for k8s"""

# Check if service is ready to handle requests

try:

# Quick Redis check

r = redis.from\_url(settings.redis\_url)

r.ping()

return {"status": "ready"}

except Exception:

return {"status": "not ready"}, 503

**Development Phases## Backend Implementation Details**

**1. Service Layer Architecture**

**1.1 Climate Data Service Structure**

class ClimateDataService:

def \_\_init\_\_(self):

# Redis initialization with fallback

try:

self.redis\_client = redis.from\_url(settings.redis\_url)

self.redis\_client.ping() # Test connection

self.use\_cache = True

except Exception as e:

print(f"Redis not available: {e}")

self.redis\_client = None

self.use\_cache = False

self.cache\_ttl = {

'geocoding': 3600 \* 24 \* 7, # 7 days

'historical': 3600 \* 24 \* 30, # 30 days

'current': 3600, # 1 hour

'projections': 3600 \* 24, # 24 hours

'full\_analysis': 3600 \* 6 # 6 hours

}

**1.2 Cache Key Patterns**

# Consistent cache key generation

def get\_cache\_key(prefix: str, \*args) -> str:

"""Generate consistent cache keys"""

# Normalize location names

normalized\_args = []

for arg in args:

if isinstance(arg, str):

# Remove extra spaces, lowercase

normalized = ' '.join(arg.lower().split())

normalized\_args.append(normalized)

elif isinstance(arg, float):

# Round coordinates to 2 decimal places

normalized\_args.append(f"{arg:.2f}")

else:

normalized\_args.append(str(arg))

return f"{prefix}:{':'.join(normalized\_args)}"

# Usage examples

cache\_key = get\_cache\_key("geocoding", "London, UK") # "geocoding:london, uk"

cache\_key = get\_cache\_key("historical", 51.5074, -0.1278) # "historical:51.51:-0.13"

**1.3 Fallback Data Generation**

def \_generate\_realistic\_current\_data(self, location\_name: str,

latitude: float, longitude: float) -> dict:

"""Generate realistic fallback data based on geography"""

# Base temperature by latitude

if abs(latitude) > 60: # Arctic/Antarctic

base\_temp = -5

elif abs(latitude) > 45: # Temperate

base\_temp = 10

elif abs(latitude) > 23.5: # Subtropical

base\_temp = 20

else: # Tropical

base\_temp = 25

# Seasonal adjustment (Northern hemisphere July is summer)

current\_month = datetime.now().month

if latitude > 0: # Northern hemisphere

if current\_month in [6, 7, 8]: # Summer

seasonal\_adjustment = 8

elif current\_month in [12, 1, 2]: # Winter

seasonal\_adjustment = -8

else:

seasonal\_adjustment = 0

else: # Southern hemisphere (reversed seasons)

if current\_month in [6, 7, 8]: # Winter

seasonal\_adjustment = -8

elif current\_month in [12, 1, 2]: # Summer

seasonal\_adjustment = 8

else:

seasonal\_adjustment = 0

current\_temp = base\_temp + seasonal\_adjustment

# Add some geographic variation

if "coast" in location\_name.lower() or "sea" in location\_name.lower():

current\_temp += 2 # Coastal moderation

if "mountain" in location\_name.lower() or "highland" in location\_name.lower():

current\_temp -= 5 # Altitude cooling

return {

"current\_temperature": round(current\_temp + random.uniform(-2, 2), 1),

"current\_humidity": round(60 + random.uniform(-20, 20)),

"avg\_temp\_max": round(current\_temp + 5, 1),

"avg\_temp\_min": round(current\_temp - 5, 1),

"total\_precipitation": round(50 + random.uniform(-30, 50), 1)

}

**2. Data Processing**

**2.1 Monthly Baseline Calculations**

def \_calculate\_monthly\_baselines(self, daily\_data: dict) -> dict:

"""Calculate monthly averages from daily data"""

import pandas as pd

# Create DataFrame for easier processing

df = pd.DataFrame({

'date': pd.to\_datetime(daily\_data['time']),

'temp\_max': daily\_data['temperature\_2m\_max'],

'temp\_min': daily\_data['temperature\_2m\_min'],

'precipitation': daily\_data['precipitation\_sum']

})

# Add month column

df['month'] = df['date'].dt.month

# Group by month and calculate averages

monthly\_avg = df.groupby('month').agg({

'temp\_max': ['mean', 'std', 'count'],

'temp\_min': ['mean', 'std', 'count'],

'precipitation': ['mean', 'sum', 'count']

}).round(2)

# Convert to dictionary format

result = {}

for month in range(1, 13):

if month in monthly\_avg.index:

month\_data = monthly\_avg.loc[month]

result[month] = {

'temp\_max\_avg': float(month\_data['temp\_max']['mean']),

'temp\_min\_avg': float(month\_data['temp\_min']['mean']),

'precipitation\_avg': float(month\_data['precipitation']['mean']),

'precipitation\_total': float(month\_data['precipitation']['sum']),

'sample\_size': int(month\_data['temp\_max']['count'])

}

return result

**2.2 Resilience Score Calculation**

def \_calculate\_resilience\_score(self, current\_data: dict, projections: dict,

variations: dict) -> int:

"""Calculate location resilience score (0-100)"""

# Base score

score = 100.0

# Temperature change impact (40% weight)

temp\_change = projections.get('temperature\_change\_2050', 0)

if temp\_change < 1.5:

temp\_penalty = 0

elif temp\_change < 2.5:

temp\_penalty = 15

elif temp\_change < 3.5:

temp\_penalty = 30

else:

temp\_penalty = 40

# Extreme weather impact (30% weight)

extreme\_days\_increase = (projections.get('extreme\_heat\_days\_future', 0) -

projections.get('extreme\_heat\_days\_current', 0))

if extreme\_days\_increase < 5:

extreme\_penalty = 0

elif extreme\_days\_increase < 15:

extreme\_penalty = 10

elif extreme\_days\_increase < 25:

extreme\_penalty = 20

else:

extreme\_penalty = 30

# Precipitation change impact (30% weight)

precip\_change = abs(projections.get('precipitation\_change\_percent', 0))

if precip\_change < 10:

precip\_penalty = 0

elif precip\_change < 20:

precip\_penalty = 10

elif precip\_change < 30:

precip\_penalty = 20

else:

precip\_penalty = 30

# Calculate final score

total\_penalty = temp\_penalty + extreme\_penalty + precip\_penalty

score = max(0, min(100, score - total\_penalty))

return int(round(score))

**3. API Integration Patterns**

**3.1 Concurrent API Calls**

async def get\_comprehensive\_climate\_analysis(self, location\_name: str) -> dict:

"""Fetch all data concurrently for performance"""

# Check cache first

cache\_key = get\_cache\_key("full\_analysis", location\_name)

if self.use\_cache:

cached = await self.\_get\_from\_cache(cache\_key)

if cached:

return cached

# Get coordinates first (required for other calls)

location\_data = await self.get\_location\_coordinates(location\_name)

if not location\_data:

return None

lat, lon = location\_data['latitude'], location\_data['longitude']

# Fetch all data concurrently

tasks = [

self.get\_current\_climate(lat, lon),

self.get\_historical\_climate\_baseline(lat, lon),

self.get\_climate\_projections(lat, lon),

self.get\_recent\_climate\_average(lat, lon)

]

try:

results = await asyncio.gather(\*tasks, return\_exceptions=True)

current, historical, projections, recent = results

# Handle any failed requests

for i, result in enumerate(results):

if isinstance(result, Exception):

print(f"Task {i} failed: {result}")

# Use fallback data

if i == 0: # Current climate

current = self.\_generate\_realistic\_current\_data(#

**Frontend Dependencies (JavaScript/TypeScript)**:

{

"dependencies": {

"next": "^14.0.0",

"react": "^18.2.0",

"react-dom": "^18.2.0",

"typescript": "^5.0.0",

"tailwindcss": "^3.3.0",

"lucide-react": "^0.263.1",

"chart.js": "^4.4.0",

"react-chartjs-2": "^5.2.0",

"axios": "^1.6.0",

"date-fns": "^2.30.0",

"react-hook-form": "^7.48.0",

"zod": "^3.22.0"

},

"devDependencies": {

"@types/node": "^20.0.0",

"@types/react": "^18.2.0",

"eslint": "^8.50.0",

"eslint-config-next": "^14.0.0",

"prettier": "^3.0.0",

"@testing-library/react": "^14.0.0",

"jest": "^29.5.0",

"jest-environment-jsdom": "^29.5.0"

}

}

**3.3 Coding Standards**

**Python Coding Standards**:

* PEP 8 compliance enforced with Black formatter
* Type hints required for all function parameters and returns
* Docstrings required for all public functions (Google style)
* Maximum line length: 88 characters (Black default)
* Import sorting with isort
* Async/await for all I/O operations

**TypeScript/JavaScript Standards**:

* ESLint with Next.js recommended config
* Prettier for consistent formatting
* TypeScript strict mode enabled
* No use of any type without explicit comment
* Functional components with hooks (no class components)
* Props interfaces defined for all components

**Git Standards**:

* Conventional commits format
* Branch names: type/ticket-description (e.g., feature/CLI-123-add-export)
* PR requires at least one review
* All tests must pass before merge
* Squash merge to main branch

**4. Environment Setup Details**

**4.1 Local Development Setup**

**Step 1: System Prerequisites**

# macOS

brew install python@3.9 node@18 postgresql@13 redis git

# Ubuntu/Debian

sudo apt update

sudo apt install python3.9 python3.9-venv nodejs npm postgresql-13 redis-server git

# Windows (use WSL2 or install individually)

# - Python 3.9+ from python.org

# - Node.js 18+ from nodejs.org

# - PostgreSQL from postgresql.org

# - Redis from Redis Windows fork

**Step 2: Repository Setup**

# Clone repository

git clone https://github.com/yourorg/climate-adaptation-app.git

cd climate-adaptation-app

# Create feature branch

git checkout -b feature/initial-setup

**Step 3: Backend Environment**

# Navigate to backend

cd backend

# Create virtual environment

python3.9 -m venv venv

# Activate virtual environment

# macOS/Linux:

source venv/bin/activate

# Windows:

venv\Scripts\activate

# Install dependencies

pip install -r requirements.txt

pip install -r requirements-dev.txt

# Create .env file

cp .env.example .env

# Edit .env with your settings:

# ENVIRONMENT=development

# LOG\_LEVEL=DEBUG

# DATABASE\_URL=postgresql://user:pass@localhost/climate\_app

# REDIS\_URL=redis://localhost:6379

# CORS\_ORIGINS=["http://localhost:3000"]

# SECRET\_KEY=your-secret-key-here

# Initialize database

python -m app.db.init

# Run migrations

alembic upgrade head

# Start backend server

uvicorn app.main:app --reload --port 8000

**Step 4: Frontend Environment**

# In new terminal, navigate to frontend

cd frontend

# Install dependencies

npm install

# Create environment file

cp .env.example .env.local

# Edit .env.local:

# NEXT\_PUBLIC\_API\_URL=http://localhost:8000

# NEXT\_PUBLIC\_ENVIRONMENT=development

# Start development server

npm run dev

# Access at http://localhost:3000

**4.2 Docker Development Setup**

**Step 1: Install Docker**

* Download Docker Desktop from docker.com
* Ensure Docker Compose is included (usually is)

**Step 2: Environment Configuration**

# Copy environment templates

cp .env.example .env

cp frontend/.env.example frontend/.env.local

cp backend/.env.example backend/.env

# Edit .env files as needed

**Step 3: Build and Run**

# Build all services

docker-compose build

# Start all services

docker-compose up

# Or run in background

docker-compose up -d

# View logs

docker-compose logs -f

# Stop services

docker-compose down

**Step 4: Verify Setup**

# Check backend health

curl http://localhost:8000/health

# Check frontend

open http://localhost:3000

# Check Redis

redis-cli ping

# Check PostgreSQL

psql -h localhost -U postgres -d climate\_app -c "SELECT 1"

**4.3 IDE Configuration**

**VS Code Setup**:

// .vscode/settings.json

{

"python.linting.enabled": true,

"python.linting.pylintEnabled": false,

"python.linting.flake8Enabled": true,

"python.formatting.provider": "black",

"python.linting.mypyEnabled": true,

"editor.formatOnSave": true,

"editor.codeActionsOnSave": {

"source.organizeImports": true

},

"[typescript]": {

"editor.defaultFormatter": "esbenp.prettier-vscode"

},

"[typescriptreact]": {

"editor.defaultFormatter": "esbenp.prettier-vscode"

}

}

**Recommended Extensions**:

* Python (ms-python.python)
* Pylance (ms-python.vscode-pylance)
* ESLint (dbaeumer.vscode-eslint)
* Prettier (esbenp.prettier-vscode)
* GitLens (eamodio.gitlens)
* Docker (ms-azuretools.vscode-docker)

**Git Repository Structure**

climate-adaptation-app/

├── .github/

│ ├── workflows/

│ │ ├── backend-ci.yml

│ │ ├── frontend-ci.yml

│ │ └── deploy.yml

│ └── ISSUE\_TEMPLATE/

├── backend/

│ ├── app/

│ │ ├── api/

│ │ ├── core/

│ │ ├── models/

│ │ ├── services/

│ │ └── main.py

│ ├── tests/

│ ├── migrations/

│ ├── requirements.txt

│ └── Dockerfile

├── frontend/

│ ├── src/

│ │ ├── app/

│ │ ├── components/

│ │ ├── hooks/

│ │ ├── lib/

│ │ └── types/

│ ├── public/

│ ├── tests/

│ ├── package.json

│ └── Dockerfile

├── docs/

│ ├── api/

│ ├── deployment/

│ └── user-guide/

├── scripts/

│ ├── setup.sh

│ ├── test.sh

│ └── deploy.sh

├── .gitignore

├── README.md

├── CHANGELOG.md

├── CONTRIBUTING.md

└── docker-compose.yml

**Development Milestones**

**Milestone 1: Foundation & Setup (Week 1-2)**

* [ ] Initialize Git repository with proper .gitignore
* [ ] Set up project structure and scaffolding
* [ ] Configure development environment (Docker, env files)
* [ ] Set up CI/CD pipelines
* [ ] Create base FastAPI application
* [ ] Create base Next.js application
* [ ] Implement basic health check endpoints
* [ ] Set up logging and error handling
* [ ] Configure CORS and security middleware
* [ ] Write initial README with setup instructions

**Deliverables**: Working development environment, basic app skeleton

**Milestone 2: Data Integration (Week 3-5)**

* [ ] Implement geocoding service with Open-Meteo
* [ ] Create climate data service (current, historical, projections)
* [ ] Integrate environmental APIs (air quality, water, biodiversity)
* [ ] Implement economic data services (World Bank, IMF)
* [ ] Add social metrics integration (WHO, UNESCO)
* [ ] Create governance data services (democracy, corruption indices)
* [ ] Implement caching layer with Redis
* [ ] Add data validation and error handling
* [ ] Create fallback data generation
* [ ] Write unit tests for all services

**Deliverables**: Complete data integration layer with 20+ sources

**Milestone 3: Backend API Development (Week 6-7)**

* [ ] Implement /locations/search endpoint
* [ ] Create /locations/analyze endpoint
* [ ] Add /locations/compare endpoint
* [ ] Implement comprehensive scoring algorithm
* [ ] Add weighted scoring calculations
* [ ] Create data aggregation pipeline
* [ ] Implement rate limiting
* [ ] Add request validation
* [ ] Create API documentation (OpenAPI)
* [ ] Write integration tests

**Deliverables**: Complete REST API with all endpoints

**Milestone 4: Frontend Core Features (Week 8-10)**

* [ ] Implement location search with autocomplete
* [ ] Create location analysis view
* [ ] Build metric cards for each category
* [ ] Implement data visualisations (charts, graphs)
* [ ] Add comparison view with radar chart
* [ ] Create weight customisation interface
* [ ] Implement progressive data loading
* [ ] Add loading states and skeletons
* [ ] Create error handling UI
* [ ] Write component tests

**Deliverables**: Functional frontend with core features

**Milestone 5: User Experience & Polish (Week 11-12)**

* [ ] Implement responsive design for mobile
* [ ] Add animations and transitions
* [ ] Create data export functionality (PDF/CSV)
* [ ] Add tooltips and help text
* [ ] Implement keyboard navigation
* [ ] Add accessibility features (ARIA labels, focus management)
* [ ] Create user preferences storage
* [ ] Add share functionality
* [ ] Implement print-friendly views
* [ ] Conduct usability testing

**Deliverables**: Polished, accessible user interface

**Milestone 6: Testing & Optimization (Week 13-14)**

* [ ] Achieve 80%+ test coverage
* [ ] Perform load testing
* [ ] Optimize API response times
* [ ] Implement query optimization
* [ ] Add performance monitoring
* [ ] Conduct security audit
* [ ] Fix identified bugs
* [ ] Optimize bundle size
* [ ] Add error tracking (Sentry)
* [ ] Create performance benchmarks

**Deliverables**: Production-ready, optimized application

**Milestone 7: Documentation & Deployment (Week 15-16)**

* [ ] Write comprehensive README
* [ ] Create API documentation
* [ ] Write user guide
* [ ] Document deployment process
* [ ] Create troubleshooting guide
* [ ] Add inline code documentation
* [ ] Set up production environment
* [ ] Configure monitoring and alerts
* [ ] Create backup strategies
* [ ] Tag v1.0.0 release

**Deliverables**: Deployed application with complete documentation

**Git Workflow & Best Practices**

**Branch Strategy**

main # Production-ready code

├── develop # Integration branch

├── feature/\* # New features

├── bugfix/\* # Bug fixes

├── hotfix/\* # Emergency fixes

└── release/\* # Release preparation

**Commit Convention**

type(scope): subject

body

footer

# Examples:

feat(api): add location comparison endpoint

fix(frontend): resolve autocomplete dropdown positioning

docs(readme): update installation instructions

test(services): add climate service unit tests

**Code Review Checklist**

* [ ] Code follows project coding standards
* [ ] All tests pass
* [ ] New code has appropriate test coverage
* [ ] Documentation is updated
* [ ] No console.logs or debugging code
* [ ] Security considerations addressed
* [ ] Performance impact considered
* [ ] Accessibility requirements met

**Documentation Requirements**

**README.md Structure**

# Climate Adaptation App

## Overview

Brief description and key features

## Quick Start

Step-by-step setup instructions

## Installation

### Prerequisites

### Backend Setup

### Frontend Setup

### Docker Setup

## Usage

### Searching Locations

### Viewing Analysis

### Comparing Locations

### Customizing Weights

## API Documentation

Link to full API docs

## Configuration

### Environment Variables

### External APIs

## Development

### Running Tests

### Code Style

### Contributing

## Troubleshooting

Common issues and solutions

## License

**API Documentation**

* OpenAPI/Swagger specification
* Authentication details
* Rate limiting information
* Request/response examples
* Error codes and meanings
* Webhook documentation (if applicable)

**Code Documentation**

def calculate\_climate\_score(data: ClimateData) -> float:

"""

Calculate climate resilience score for a location.

Args:

data: Climate data including temperature, precipitation,

and extreme weather metrics

Returns:

Float score between 0-100, where higher is better

Raises:

ValueError: If required data fields are missing

Example:

>>> score = calculate\_climate\_score(london\_data)

>>> print(f"Climate score: {score}")

72.5

"""

**Quality Assurance Checklist**

**Pre-Release Checklist**

* [ ] All milestones completed
* [ ] Test coverage > 80%
* [ ] No critical or high severity bugs
* [ ] Performance benchmarks met
* [ ] Security audit passed
* [ ] Documentation complete
* [ ] Accessibility audit passed
* [ ] Cross-browser testing complete
* [ ] Load testing successful
* [ ] Monitoring configured

**Release Process**

1. Create release branch from develop
2. Update version numbers
3. Update CHANGELOG.md
4. Run full test suite
5. Deploy to staging
6. Conduct final QA
7. Merge to main
8. Tag release
9. Deploy to production
10. Monitor for issues

**Maintenance & Updates**

**Regular Tasks**

* **Daily**: Monitor error logs and performance metrics
* **Weekly**: Review and triage issues, update dependencies
* **Monthly**: Security patches, data source validation
* **Quarterly**: Major dependency updates, performance review

**Changelog Format**

# Changelog

## [1.1.0] - 2024-02-15

### Added

- Multi-language support for 10 languages

- Historical trend charts for all metrics

### Changed

- Improved scoring algorithm accuracy

- Updated to Next.js 14.1

### Fixed

- Memory leak in data aggregation service

- Incorrect timezone handling for Pacific islands

### Security

- Updated dependencies to patch CVE-2024-XXXX

**Technical Architecture**

**Technology Stack**

**Backend**

* **Framework**: FastAPI (Python 3.9+)
* **API Style**: RESTful with async/await support
* **Data Processing**: Pandas for climate data analysis
* **Caching**: Redis (optional, with graceful fallback)
* **Database**: PostgreSQL with PostGIS (future enhancement)
* **Deployment**: Railway.app with environment-based configuration

**Frontend**

* **Framework**: Next.js 14 with TypeScript
* **UI Library**: React 18.2
* **Styling**: Tailwind CSS 3.3
* **Icons**: Lucide React
* **State Management**: React hooks (useState, useEffect)
* **API Communication**: Native fetch with async/await

**Infrastructure**

* **Backend Hosting**: Railway.app
* **Frontend Hosting**: Vercel
* **Domain**: climate-migration-app.openeyemedia.net
* **SSL**: Automatic via hosting providers
* **Monitoring**: Console logging with structured output

**Functional Requirements**

**1. Location Search & Geocoding**

**1.1 Search Functionality**

* **Autocomplete**: Real-time location suggestions as user types (minimum 2 characters)
* **Search API**: Integration with Open-Meteo Geocoding API
* **Results Display**: Show city, region/state, country, and population
* **Debouncing**: 300ms delay to optimise API calls
* **Result Limit**: Maximum 8 suggestions per search

**1.2 Location Data Structure**

interface LocationOption {

name: string;

country: string;

admin1?: string;

latitude: number;

longitude: number;

population?: number;

timezone?: string;

display\_name: string;

}

**2. Climate Data Analysis**

**2.1 Current Climate Metrics**

* **Temperature**: Current temperature with proper formatting (1 decimal place + °C)
* **Humidity**: Current relative humidity percentage
* **Precipitation**: Total monthly/annual precipitation
* **Temperature Range**: Average max/min temperatures

**2.2 Climate Change Indicators**

* **Monthly Variations**: Temperature and rainfall changes vs 1990-2020 baseline
* **Annual Temperature Increase**: 5-year average vs 30-year baseline
* **Data Quality Indicators**: "Real" vs "Estimated" data confidence levels
* **Seasonal Patterns**: Month-specific analysis (currently July-focused)

**2.3 Future Projections (2050)**

* **Temperature Change**: Projected increase by 2050
* **Extreme Heat Days**: Current vs future frequency
* **Precipitation Changes**: Percentage change in rainfall patterns
* **Risk Assessment**: Human-readable risk levels and descriptions

**3. Resilience Scoring Algorithm**

**3.1 Score Components**

* **Temperature Stability**: Weight changes in average temperatures
* **Extreme Weather**: Factor in heat waves and temperature variability
* **Precipitation Patterns**: Consider drought and flood risks
* **Geographic Factors**: Latitude-based adjustments

**3.2 Score Calculation**

resilience\_score = 100 - (

temp\_change\_impact \* 0.4 +

extreme\_weather\_impact \* 0.3 +

precipitation\_impact \* 0.3

)

**3. Risk Assessment**

**3.1 Risk Levels**

* **Low Risk**: Minimal climate change impacts expected
* **Moderate Risk**: Some adaptation needed
* **High Risk**: Significant climate challenges
* **Very High Risk**: Major climate threats

**3.2 Risk Calculation Logic**

def calculate\_risk\_level(temp\_change: float, extreme\_days\_increase: int,

precip\_change: float) -> tuple[str, str]:

"""

Returns (risk\_level, description)

"""

# Temperature change thresholds

if temp\_change < 1.5:

temp\_risk = 1

elif temp\_change < 2.5:

temp\_risk = 2

elif temp\_change < 3.5:

temp\_risk = 3

else:

temp\_risk = 4

# Extreme heat days increase

if extreme\_days\_increase < 10:

heat\_risk = 1

elif extreme\_days\_increase < 20:

heat\_risk = 2

elif extreme\_days\_increase < 30:

heat\_risk = 3

else:

heat\_risk = 4

# Precipitation change (both increase and decrease are risky)

precip\_abs = abs(precip\_change)

if precip\_abs < 10:

precip\_risk = 1

elif precip\_abs < 20:

precip\_risk = 2

elif precip\_abs < 30:

precip\_risk = 3

else:

precip\_risk = 4

# Overall risk is weighted average

overall\_risk = (temp\_risk \* 0.4 + heat\_risk \* 0.4 + precip\_risk \* 0.2)

if overall\_risk < 1.5:

return "Low", "This location shows minimal climate change impacts"

elif overall\_risk < 2.5:

return "Moderate", "Some climate adaptation measures will be needed"

elif overall\_risk < 3.5:

return "High", "Significant climate challenges expected"

else:

return "Very High", "Major climate threats require extensive adaptation"

**3.3 Key Concerns Generation**

def generate\_key\_concerns(location\_data: dict) -> list[str]:

concerns = []

# Temperature concerns

if location\_data["projections"]["temperature\_change\_2050"] > 2.5:

concerns.append("Extreme heat waves will become more frequent")

# Precipitation concerns

precip\_change = location\_data["projections"]["precipitation\_change\_percent"]

if precip\_change < -20:

concerns.append("Increased drought risk")

elif precip\_change > 20:

concerns.append("Increased flooding risk")

# Coastal concerns (if applicable)

if location\_data["location"]["latitude"] < 60 and is\_coastal(location\_data):

concerns.append("Sea level rise impacts")

# Infrastructure concerns

if location\_data["projections"]["extreme\_heat\_days\_future"] > 30:

concerns.append("Infrastructure strain from extreme heat")

return concerns[:3] # Return top 3 concerns

**5. Recommendations Engine**

**5.1 Recommendation Categories**

* **Infrastructure**: Climate-adapted building needs
* **Water Management**: Conservation and storage
* **Energy**: Renewable energy adoption
* **Community**: Climate preparedness planning
* **Economic**: Climate-resilient industries

**5.2 Personalisation (Future)**

* User climate preferences
* Economic constraints
* Family considerations
* Career compatibility

**Data Requirements**

**1. External Data Sources**

**1.1 Climate & Weather Data**

* **Open-Meteo Suite**: Weather, climate projections, historical data
* **NOAA Climate Data**: US-specific detailed climate records
* **ERA5 Reanalysis**: European climate reanalysis data
* **World Bank Climate Portal**: Country-level climate indicators

**1.2 Environmental Data**

* **Global Forest Watch API**: Deforestation and forest cover
* **IUCN Red List API**: Biodiversity and species data
* **Water.org API**: Water stress and quality metrics
* **OpenAQ API**: Air quality measurements globally

**1.3 Energy & Net Zero Data**

* **IEA Data Services**: Energy mix and transition metrics
* **Global Carbon Atlas**: Emissions data by country
* **IRENA Statistics**: Renewable energy deployment
* **EV-Volumes.com**: Electric vehicle adoption rates

**1.4 Economic Data**

* **World Bank Open Data**: GDP, inflation, employment
* **IMF Data API**: Economic indicators and forecasts
* **Numbeo API**: Cost of living indices
* **OECD Stats**: Income inequality, labour markets

**1.5 Social & Governance Data**

* **WHO Global Health Observatory**: Healthcare metrics
* **UNESCO Institute for Statistics**: Education data
* **Transparency International**: Corruption indices
* **Freedom House API**: Democracy scores
* **World Happiness Report**: Wellbeing rankings

**1.6 Demographics & Food**

* **UN Population Division**: Demographics projections
* **FAO Statistics**: Food security and agriculture
* **World Food Programme**: Food insecurity data
* **Migration Data Portal**: Migration flows

**2. Data Integration Architecture**

**2.1 Data Pipeline**

class DataAggregator:

def \_\_init\_\_(self):

self.sources = {

'climate': ClimateDataService(),

'environment': EnvironmentDataService(),

'energy': EnergyDataService(),

'economy': EconomyDataService(),

'social': SocialDataService(),

'governance': GovernanceDataService(),

'demographics': DemographicsDataService(),

'happiness': HappinessDataService()

}

async def get\_location\_analysis(self, location: str) -> dict:

"""Aggregate data from all sources"""

tasks = []

for category, service in self.sources.items():

tasks.append(service.get\_data(location))

results = await asyncio.gather(\*tasks, return\_exceptions=True)

return self.combine\_results(results)

**2.2 Data Schema**

interface LocationAnalysis {

location: LocationData;

climate: ClimateMetrics;

environment: EnvironmentMetrics;

energy: EnergyTransitionMetrics;

economy: EconomicIndicators;

social: SocialWellbeingMetrics;

governance: GovernanceMetrics;

demographics: DemographicData;

food: FoodSecurityMetrics;

happiness: HappinessIndicators;

overall\_score: number;

comparative\_ranking: number;

}

interface ClimateMetrics {

current\_conditions: CurrentWeather;

climate\_trends: ClimateChangeIndicators;

extreme\_events: ExtremeWeatherFrequency;

future\_projections: ClimateProjections;

adaptation\_progress: AdaptationMetrics;

}

interface EnvironmentMetrics {

biodiversity: BiodiversityIndicators;

water: WaterResourceMetrics;

air\_quality: AirQualityData;

land\_use: LandUseChange;

pollution: PollutionLevels;

}

interface EnergyTransitionMetrics {

renewable\_share: number;

carbon\_intensity: number;

ev\_adoption: number;

energy\_efficiency: number;

policy\_support: PolicyIndicators;

}

interface EconomicIndicators {

gdp\_per\_capita: number;

inflation\_rate: number;

unemployment: UnemploymentData;

cost\_of\_living: CostOfLivingIndex;

income\_inequality: number;

economic\_stability: StabilityScore;

}

interface SocialWellbeingMetrics {

healthcare: HealthcareQuality;

education: EducationMetrics;

safety: SafetyIndicators;

social\_support: SocialSafetyNet;

discrimination: DiscriminationLevels;

}

**3. Data Caching Strategy**

**3.1 Cache Durations by Data Type**

CACHE\_DURATIONS = {

# Real-time data (1 hour)

'current\_weather': 3600,

'air\_quality': 3600,

# Daily updates (24 hours)

'climate\_projections': 86400,

'energy\_mix': 86400,

'covid\_stats': 86400,

# Weekly updates (7 days)

'economic\_indicators': 604800,

'biodiversity': 604800,

'water\_quality': 604800,

# Monthly updates (30 days)

'governance\_scores': 2592000,

'happiness\_index': 2592000,

'demographics': 2592000,

# Annual updates (365 days)

'democracy\_index': 31536000,

'corruption\_index': 31536000

}

**3.2 Progressive Data Loading**

// Load data in priority order

const loadLocationData = async (location: string) => {

// Phase 1: Critical data (immediate)

const critical = await Promise.all([

fetchClimateData(location),

fetchBasicEconomics(location)

]);

updateUI(critical);

// Phase 2: Important data (1-2 seconds)

const important = await Promise.all([

fetchEnvironmentData(location),

fetchEnergyData(location),

fetchSocialData(location)

]);

updateUI(important);

// Phase 3: Supplementary data (2-5 seconds)

const supplementary = await Promise.all([

fetchGovernanceData(location),

fetchDemographicsData(location),

fetchHappinessData(location)

]);

updateUI(supplementary);

};

**4. Data Quality & Validation**

**4.1 Source Reliability Scoring**

SOURCE\_RELIABILITY = {

'world\_bank': 0.95,

'un\_agencies': 0.95,

'oecd': 0.90,

'government\_stats': 0.85,

'ngo\_reports': 0.80,

'crowdsourced': 0.70

}

def calculate\_data\_confidence(sources: list[str]) -> float:

"""Calculate confidence score based on data sources"""

if not sources:

return 0.0

scores = [SOURCE\_RELIABILITY.get(s, 0.5) for s in sources]

return sum(scores) / len(scores)

**4.2 Missing Data Handling**

class DataImputation:

@staticmethod

def impute\_missing(data: dict, location: Location) -> dict:

"""Intelligently fill missing data"""

# Regional averages for missing country data

if not data.get('democracy\_index'):

data['democracy\_index'] = RegionalAverages.get\_democracy(

location.region

)

# Correlation-based imputation

if not data.get('corruption\_index') and data.get('democracy\_index'):

# Strong inverse correlation

data['corruption\_index'] = DataImputation.estimate\_corruption(

data['democracy\_index']

)

# Time-series extrapolation

if not data.get('current\_year\_data'):

data['current\_year\_data'] = DataImputation.extrapolate(

data['historical\_data']

)

return data

**Non-Functional Requirements**

**1. Performance**

**1.1 Response Times**

* **Search Autocomplete**: <500ms
* **Climate Analysis**: <3 seconds
* **Page Load**: <2 seconds
* **API Response**: <1 second for cached data

**1.2 Scalability**

* **Concurrent Users**: Support 1000+ simultaneous users
* **API Rate Limits**: Respect third-party API limits
* **Caching**: Reduce external API calls by 80%

**2. Reliability**

**2.1 Availability**

* **Uptime Target**: 99.5%
* **Graceful Degradation**: Function without Redis
* **Fallback Data**: Always provide estimates if APIs fail
* **Error Recovery**: Automatic retry with exponential backoff

**2.2 Data Consistency**

* **Cache Coherence**: Same location = same data for cache duration
* **Version Control**: Track data schema versions
* **Audit Trail**: Log all data source accesses

**3. Security**

**3.1 API Security**

* **CORS**: Restrictive origin policies
* **HTTPS**: Enforced for all communications
* **Rate Limiting**: Prevent API abuse
* **Input Validation**: Sanitise all user inputs

**3.2 Data Privacy**

* **No Personal Data**: App doesn't collect user information
* **Anonymous Usage**: No tracking or analytics
* **Secure Communications**: TLS 1.2+ for all APIs

**4. Usability**

**4.1 User Interface**

* **Responsive Design**: Mobile-first approach
* **Accessibility**: WCAG 2.1 AA compliance
* **Loading States**: Clear feedback during data fetching
* **Error Messages**: User-friendly error explanations

**4.2 Data Presentation**

* **Visual Hierarchy**: Most important data prominent
* **Consistent Formatting**: Temperature always "XX.X°C"
* **Colour Coding**: Risk levels with intuitive colours
* **Progressive Disclosure**: Details on demand

**API Specifications**

**1. Backend Endpoints**

**1.1 Core Endpoints**

GET / # API information

GET /health # Health check

GET /locations/search # Location autocomplete

POST /climate/analyze # Comprehensive analysis

**1.2 Request/Response Formats**

**Location Search**

// Request

GET /locations/search?q={query}&limit={limit}

// Success Response (200)

{

"success": true,

"locations": [

{

"name": "London",

"country": "United Kingdom",

"admin1": "England",

"latitude": 51.5074,

"longitude": -0.1278,

"population": 8982000,

"timezone": "Europe/London",

"display\_name": "London, England, United Kingdom"

}

],

"query": "london"

}

// Error Response (200) - Note: Always returns 200, check success flag

{

"success": false,

"error": "Error message",

"locations": []

}

**Climate Analysis**

// Request

POST /climate/analyze

Content-Type: application/json

{

"location": "London, UK" // Free-form location string

}

// Success Response (200)

{

"success": true,

"data": {

"location": {

"name": "London",

"country": "United Kingdom",

"latitude": 51.5074,

"longitude": -0.1278,

"population": 8982000,

"timezone": "Europe/London"

},

"current\_climate": {

"current\_temperature": 15.4, // Always 1 decimal place

"current\_humidity": 76, // Integer percentage

"avg\_temp\_max": 18.2, // Monthly average max

"avg\_temp\_min": 11.3, // Monthly average min

"total\_precipitation": 45.2 // Monthly total mm

},

"climate\_variations": {

"current\_month": 7, // 1-12

"month\_name": "July",

"temp\_max\_variation": 2.1, // °C change vs baseline

"temp\_min\_variation": 1.8, // °C change vs baseline

"rainfall\_variation\_percent": -15, // % change vs baseline

"baseline\_period": "1990-2020",

"recent\_period": "2020-2024",

"data\_quality": "Real" // "Real" or "Estimated"

},

"annual\_temp\_increase": {

"increase": 1.2, // °C increase

"recent\_avg": 11.4, // Recent 5-year average

"baseline\_avg": 10.2, // 30-year baseline average

"baseline\_period": "1990-2020",

"recent\_period": "2020-2024",

"confidence": "High" // "High", "Medium", "Low"

},

"projections": {

"temperature\_change\_2050": 2.8, // °C increase by 2050

"current\_avg\_temp": 10.5,

"future\_avg\_temp": 13.3,

"extreme\_heat\_days\_current": 5, // Days >30°C per year

"extreme\_heat\_days\_future": 18,

"precipitation\_change\_percent": -12 // % change by 2050

},

"resilience\_score": 72, // 0-100 scale

"risk\_assessment": {

"risk\_level": "Moderate", // Low/Moderate/High/Very High

"description": "London faces moderate climate risks...",

"temperature\_impact": "Moderate warming expected",

"key\_concerns": [

"Increased flooding risk",

"Summer heat waves",

"Infrastructure strain"

]

},

"recommendations": [

"Invest in flood defenses",

"Upgrade cooling systems",

"Implement water conservation"

]

}

}

// Error Response (200)

{

"success": false,

"error": "Could not find climate data for location: InvalidPlace"

}

**2. External API Integrations**

**2.1 Open-Meteo APIs**

**Geocoding API**

# Endpoint

https://geocoding-api.open-meteo.com/v1/search

# Parameters

params = {

"name": "London", # Required: location name

"count": 1, # Results limit (1-100)

"language": "en", # Language code

"format": "json" # Response format

}

# Response Structure

{

"results": [

{

"id": 2643743,

"name": "London",

"latitude": 51.50853,

"longitude": -0.12574,

"elevation": 25.0,

"feature\_code": "PPLC",

"country\_code": "GB",

"admin1\_id": 6269131,

"admin1": "England",

"country\_id": 2635167,

"country": "United Kingdom",

"population": 8961989,

"timezone": "Europe/London"

}

]

}

**Current Weather API**

# Endpoint

https://api.open-meteo.com/v1/forecast

# Parameters

params = {

"latitude": 51.5074,

"longitude": -0.1278,

"current": ["temperature\_2m", "relative\_humidity\_2m"],

"daily": ["temperature\_2m\_max", "temperature\_2m\_min", "precipitation\_sum"],

"timezone": "auto"

}

# Response Structure

{

"current": {

"time": "2025-07-05T12:00",

"temperature\_2m": 15.4,

"relative\_humidity\_2m": 76

},

"daily": {

"time": ["2025-07-05", "2025-07-06", ...],

"temperature\_2m\_max": [18.2, 17.8, ...],

"temperature\_2m\_min": [11.3, 10.9, ...],

"precipitation\_sum": [2.1, 0.0, ...]

}

}

**Historical Climate API (Archive)**

# Endpoint

https://archive-api.open-meteo.com/v1/archive

# Parameters

params = {

"latitude": 51.5074,

"longitude": -0.1278,

"start\_date": "1990-01-01",

"end\_date": "2020-12-31",

"daily": ["temperature\_2m\_max", "temperature\_2m\_min", "precipitation\_sum"],

"timezone": "auto"

}

# Note: Large response with daily data for 30 years

# Process into monthly averages for baseline calculations

**Climate Projections API**

# Endpoint

https://climate-api.open-meteo.com/v1/climate

# Parameters

params = {

"latitude": 51.5074,

"longitude": -0.1278,

"start\_date": "2024-01-01",

"end\_date": "2050-12-31",

"models": ["CMCC\_CM2\_VHR4", "MRI\_AGCM3\_2\_S"],

"daily": ["temperature\_2m\_max", "temperature\_2m\_min", "precipitation\_sum"],

"timezone": "auto"

}

**2.2 Error Handling**

# Comprehensive error handling pattern

async def fetch\_with\_retry(url: str, params: dict, max\_retries: int = 3):

for attempt in range(max\_retries):

try:

async with httpx.AsyncClient() as client:

response = await client.get(

url,

params=params,

timeout=httpx.Timeout(30.0)

)

response.raise\_for\_status()

return response.json()

except httpx.TimeoutException:

if attempt == max\_retries - 1:

raise

await asyncio.sleep(2 \*\* attempt) # Exponential backoff

except httpx.HTTPStatusError as e:

if e.response.status\_code == 429: # Rate limit

await asyncio.sleep(60) # Wait 1 minute

elif e.response.status\_code >= 500: # Server error

if attempt < max\_retries - 1:

await asyncio.sleep(5)

continue

raise

except Exception as e:

logger.error(f"API request failed: {e}")

raise

**Frontend Implementation Details**

**1. Component Architecture**

**1.1 Main Component Structure**

// ClimateApp.tsx structure

const ClimateApp: React.FC = () => {

// State management

const [selectedLocation, setSelectedLocation] = useState<LocationOption | null>(null);

const [analysisData, setAnalysisData] = useState<ClimateAnalysis | null>(null);

const [isLoading, setIsLoading] = useState(false);

const [error, setError] = useState<string | null>(null);

// API URL configuration

const API\_BASE\_URL = process.env.NODE\_ENV === 'production'

? 'https://climate-migration-app.openeyemedia.net/api'

: 'http://localhost:8000';

return (

<div className="min-h-screen bg-gradient-to-b from-blue-50 to-green-50">

<LocationSearch {...} />

{isLoading && <LoadingState />}

{error && <ErrorState error={error} />}

{analysisData && <AnalysisResults data={analysisData} />}

</div>

);

};

**1.2 Location Search Implementation**

const LocationSearch: React.FC<LocationSearchProps> = ({

placeholder,

onLocationSelect,

selectedLocation

}) => {

const [query, setQuery] = useState('');

const [suggestions, setSuggestions] = useState<LocationOption[]>([]);

const [isLoading, setIsLoading] = useState(false);

const [showDropdown, setShowDropdown] = useState(false);

const searchRef = useRef<HTMLDivElement>(null);

const debounceTimer = useRef<NodeJS.Timeout>();

// Debounced search

useEffect(() => {

if (query.length < 2) {

setSuggestions([]);

return;

}

clearTimeout(debounceTimer.current);

debounceTimer.current = setTimeout(() => {

searchLocations(query);

}, 300);

return () => clearTimeout(debounceTimer.current);

}, [query]);

// Click outside handler

useEffect(() => {

const handleClickOutside = (event: MouseEvent) => {

if (searchRef.current && !searchRef.current.contains(event.target as Node)) {

setShowDropdown(false);

}

};

document.addEventListener('mousedown', handleClickOutside);

return () => document.removeEventListener('mousedown', handleClickOutside);

}, []);

};

**2. Data Formatting Functions**

**2.1 Temperature Formatting**

const formatTemperature = (temp: number | undefined): string => {

if (typeof temp !== 'number' || isNaN(temp)) return '--';

return `${Math.round(temp \* 10) / 10}°C`;

};

const formatTemperatureChange = (change: number | undefined): string => {

if (typeof change !== 'number' || isNaN(change)) return '--';

const sign = change >= 0 ? '+' : '';

return `${sign}${Math.round(change \* 10) / 10}°C`;

};

**2.2 Percentage Formatting**

const formatPercentage = (value: number | undefined): string => {

if (typeof value !== 'number' || isNaN(value)) return '--';

const sign = value >= 0 ? '+' : '';

return `${sign}${Math.round(value)}%`;

};

**2.3 Score Calculations for Display**

// Convert variations to 0-100 scale for progress bars

const getVariationScore = (variation: number, maxVariation: number): number => {

// Map -maxVariation to +maxVariation onto 0-100 scale

// 0°C = 50%, +max = 100%, -max = 0%

const normalized = (variation + maxVariation) / (2 \* maxVariation);

return Math.max(0, Math.min(100, Math.round(normalized \* 100)));

};

const getRainfallScore = (percentChange: number): number => {

// Map -100% to +200% onto 0-100 scale

// 0% = 33.3%, +100% = 66.7%, +200% = 100%

const normalized = (percentChange + 100) / 300;

return Math.max(0, Math.min(100, Math.round(normalized \* 100)));

};

**3. Error Handling**

**3.1 API Error Handling**

const analyzeLocation = async (location: LocationOption) => {

setIsLoading(true);

setError(null);

try {

const response = await fetch(`${API\_BASE\_URL}/climate/analyze`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ location: location.display\_name })

});

const data = await response.json();

if (!data.success) {

throw new Error(data.error || 'Failed to analyze location');

}

setAnalysisData(data.data);

} catch (err) {

setError(err instanceof Error ? err.message : 'An unexpected error occurred');

setAnalysisData(null);

} finally {

setIsLoading(false);

}

};

**3.2 User-Friendly Error Messages**

const getErrorMessage = (error: string): string => {

const errorMap: Record<string, string> = {

'Network request failed': 'Unable to connect to the server. Please check your connection.',

'Could not find climate data': 'Climate data is not available for this location.',

'Rate limit exceeded': 'Too many requests. Please try again in a minute.',

'Timeout': 'The request took too long. Please try again.'

};

return errorMap[error] || error;

};

**4. Loading States**

**4.1 Skeleton Loading**

const LoadingState: React.FC = () => (

<div className="animate-pulse">

<div className="h-8 bg-gray-200 rounded w-3/4 mb-4"></div>

<div className="h-4 bg-gray-200 rounded w-1/2 mb-2"></div>

<div className="h-4 bg-gray-200 rounded w-2/3"></div>

</div>

);

**4.2 Progressive Data Loading**

// Show data as it becomes available

const AnalysisResults: React.FC<{ data: ClimateAnalysis }> = ({ data }) => {

const [expandedSections, setExpandedSections] = useState({

current: true,

variations: false,

projections: false,

recommendations: false

});

return (

<div className="space-y-4">

{data.current\_climate && <CurrentClimateSection data={data.current\_climate} />}

{data.climate\_variations && <VariationsSection data={data.climate\_variations} />}

{data.projections && <ProjectionsSection data={data.projections} />}

{data.recommendations && <RecommendationsSection data={data.recommendations} />}

</div>

);

};

**Phase 1: MVP Enhancement (Current)**

* ✅ Real data integration
* ✅ Consistent data caching
* ✅ Climate variation metrics
* ✅ Basic risk assessment
* ⏳ Production deployment optimisation

**Phase 2: Advanced Features (Next)**

* 🔲 User accounts and preferences
* 🔲 Saved location comparisons
* 🔲 Historical trend visualisations
* 🔲 Email alerts for climate updates
* 🔲 API rate limit management

**Phase 3: Data Enrichment**

* 🔲 ND-GAIN integration
* 🔲 Economic indicators
* 🔲 Infrastructure assessments
* 🔲 Migration cost calculator
* 🔲 Community data integration

**Phase 4: Scale & Monetisation**

* 🔲 Premium data access tiers
* 🔲 Detailed PDF reports
* 🔲 Business/Enterprise APIs
* 🔲 White-label solutions
* 🔲 Partner integrations

**Testing Requirements**

**1. Unit Testing**

* **Backend**: Python pytest for services
* **Frontend**: Jest for React components
* **Coverage Target**: 80% code coverage
* **Mocking**: External API responses

**2. Integration Testing**

* **API Tests**: Full request/response cycles
* **Data Flow**: End-to-end data validation
* **Cache Testing**: Verify consistency
* **Fallback Testing**: Simulate API failures

**3. User Acceptance Testing**

* **Location Search**: Various global locations
* **Data Accuracy**: Verify against known data
* **Performance**: Load testing with concurrent users
* **Cross-browser**: Chrome, Firefox, Safari, Edge

**Deployment Requirements**

**1. Environment Configuration**

* **Development**: Local with test data
* **Staging**: Mirrors production
* **Production**: Live with monitoring

**2. Environment Variables**

# Backend

PORT=8000

REDIS\_URL=redis://...

NODE\_ENV=production

CORS\_ORIGINS=https://climate-migration-app.openeyemedia.net

# Frontend

NEXT\_PUBLIC\_API\_URL=https://climate-migration-app.openeyemedia.net/api

**3. Deployment Process**

* **Backend**: Railway.app auto-deployment from main branch
* **Frontend**: Vercel auto-deployment
* **Database Migrations**: Not yet implemented
* **Rollback Strategy**: Git revert with redeploy

**Monitoring & Maintenance**

**1. Application Monitoring**

* **Uptime Monitoring**: Check health endpoints
* **Error Tracking**: Log aggregation
* **Performance Metrics**: Response time tracking
* **API Usage**: Monitor rate limits

**2. Data Quality Monitoring**

* **Source Availability**: Track API uptime
* **Data Freshness**: Verify cache effectiveness
* **Accuracy Validation**: Periodic manual checks
* **User Feedback**: Report mechanisms

**Future Enhancements**

**1. Technical Enhancements**

* GraphQL API for flexible queries
* WebSocket for real-time updates
* Machine learning for predictions
* Blockchain for data verification

**2. Feature Enhancements**

* Multi-language support
* Voice interface
* AR visualisation
* Social features
* Migration communities

**3. Data Enhancements**

* Satellite imagery integration
* Real-time disaster alerts
* Air quality indices
* Economic indicators
* Healthcare accessibility

**Success Metrics**

**1. Technical Metrics**

* **API Response Time**: <1s average
* **Uptime**: >99.5%
* **Cache Hit Rate**: >80%
* **Error Rate**: <0.1%

**2. User Metrics**

* **Search Completion**: >90%
* **Analysis Views**: >70% of searches
* **Return Users**: >40%
* **User Satisfaction**: >4.5/5

**3. Business Metrics**

* **API Usage Growth**: 20% MoM
* **Premium Conversions**: 5%
* **Partner Integrations**: 10+
* **Revenue Growth**: 30% QoQ