Simulation of DeepCAL’s Decision Engine using Neutrosophic AHP and TOPSIS

Overview: We demonstrate DeepCAL’s core decision-making process by first using a neutrosophic Analytic Hierarchy Process (AHP) to derive criteria weights for cost, reliability, and responsiveness. We then apply the TOPSIS multi-criteria ranking method to evaluate 4 hypothetical freight forwarders on these criteria. This simulation uses Triangular Neutrosophic Numbers (TNNs) for pairwise comparisons and a simple neutrosophic score function to handle uncertainty in expert judgments. Finally, we test how changes in cost or reliability affect the rankings (sensitivity analysis).

Step 1: Neutrosophic Pairwise Judgments for Criteria Importance

We begin by gathering expert judgments on the relative importance of the three criteria. Instead of exact numeric ratios, experts provide their opinions as Triangular Neutrosophic Numbers (TNNs), which capture the truth (T), indeterminacy (I), and falsity (F) of statements like “Criterion X is more important than Criterion Y.” For example, a TNN judgment (T, I, F) = (0.7, 0.1, 0.2) for “cost vs responsiveness” indicates a high truth degree (0.7) that cost is more important than responsiveness, low falsity (0.2) for that statement, and some indeterminacy (0.1). To convert these neutrosophic judgments into a single usable score, we use the simple score function:

𝑆

(

TNN

)

=

𝑇

−

𝐹

.

S(TNN)=T−F. This yields a crisp number reflecting the net importance of one criterion over the other (ignoring indeterminacy for the moment). A positive $S$ means the first criterion is judged more important, a negative $S$ means the second is more important, and $S = 0$ means they are equally important. Table 1 shows three pairwise comparisons among cost (C), reliability (R), and responsiveness (Resp) with hypothetical neutrosophic values and their resulting scores: Table 1. Neutrosophic judgments and score for each criteria pair

Pair (Criterion i vs j) Triangular Neutrosophic Judgment (T, I, F) Score

𝑆

=

𝑇

−

𝐹

S=T−F Interpretation (crisp comparison)

Cost vs Reliability (0.3, 0.1, 0.6) $0.3 - 0.6 = -0.3$ Negative – Cost is less important than Reliability.

Cost vs Responsiveness (0.7, 0.1, 0.2) $0.7 - 0.2 = 0.5$ Positive – Cost is more important than Responsiveness.

Reliability vs Responsiveness (0.8, 0.1, 0.1) $0.8 - 0.1 = 0.7$ Positive – Reliability is more important than Responsiveness.

Each $S$ value can be mapped to a pairwise comparison scale for AHP. We interpret $1 + S$ as the relative importance of the row criterion over the column criterion (for $S\ge0$), or its reciprocal if $S$ is negative. For instance:

For Cost vs Reliability, $S=-0.3$ indicates cost is viewed as less important. We take

1

+

𝑆

=

1

−

0.3

=

0.7

1+S=1−0.3=0.7 as the Cost/Reliability comparison value. This means cost is only 0.7 times as important as reliability (i.e. reliability is more crucial). The reciprocal Reliability/Cost value becomes

1

/

0.7

≈

1.428

1/0.7≈1.428.

For Cost vs Responsiveness, $S=0.5$ gives

1

+

𝑆

=

1.5

1+S=1.5, so cost is 1.5 times as important as responsiveness. (Reciprocal Responsiveness/Cost = 1/1.5 ≈ 0.667.)

For Reliability vs Responsiveness, $S=0.7$ gives

1

+

𝑆

=

1.7

1+S=1.7, meaning reliability is 1.7 times more important than responsiveness. (Reciprocal Responsiveness/Reliability = 1/1.7 ≈ 0.588.)

Using these, we construct the crisp pairwise comparison matrix $A$ for the three criteria: Table 2. Crisp pairwise comparison matrix $A$ (derived from neutrosophic scores)

Criteria Cost Reliability Responsiveness

Cost 1.000 0.700 1.500

Reliability 1.428 1.000 1.700

Responsiveness 0.667 0.588 1.000

Explanation: Each entry $a\_{ij}$ represents how many times more important row criterion i is than column criterion j. Entries below 1 indicate lower importance. (The matrix is reciprocal, so $a\_{ji} = 1/a\_{ij}$ and all diagonal entries are 1.)

Step 2: Computing Criteria Weights (AHP)

With the comparison matrix $A$, we calculate the weight (priority) of each criterion. We use the standard AHP normalized eigenvector approach, which can be approximated by the column normalization and row averaging method:

Normalize Columns: Sum each column of $A$ and divide each entry by its column sum.

– Column sums: for Cost = 1 + 1.428 + 0.667 = 3.096; for Reliability = 0.700 + 1 + 0.588 = 2.288; for Responsiveness = 1.500 + 1.700 + 1 = 4.200.

– Normalize each entry: $n\_{ij} = a\_{ij} / (\text{column sum of j})$. For example, normalized Cost vs Reliability = $0.700 / 2.288 = 0.306$. The normalized matrix (rounded to 3 decimals) is:

𝑁

=

[

0.323

0.306

0.357

0.461

0.437

0.405

0.216

0.257

0.238

]

,

N=

​

0.323

0.461

0.216

​

0.306

0.437

0.257

​

0.357

0.405

0.238

​

​

,

where each row corresponds to Cost, Reliability, Responsiveness respectively.

Average Rows: The approximate weight for each criterion is the average of its normalized row. For example, Cost’s weight $w\_{\text{cost}} = (0.323 + 0.306 + 0.357)/3 = 0.329$ (32.9%). Calculating these averages gives the criteria weight vector $w$:

Cost: $w\_{\text{cost}} \approx 0.329$ (32.9%)

Reliability: $w\_{\text{rel}} \approx 0.434$ (43.4%)

Responsiveness: $w\_{\text{resp}} \approx 0.237$ (23.7%)

As expected, Reliability receives the highest weight (most important), followed by Cost, then Responsiveness, based on our judgments.

Step 3: Consistency Check of Judgments

In AHP, it’s important to check the consistency of the pairwise comparisons. We compute the Consistency Ratio (CR) to ensure the expert’s judgments were not random or contradictory. The steps are:

Compute the principal eigenvalue $\lambda\_{\max}$ of matrix $A$. Using $w$ as an approximation of the principal eigenvector, we find $\lambda\_{\max}$ by $A w$ and dividing by $w$ element-wise. For our matrix, $\lambda\_{\max} \approx 3.006$.

The Consistency Index (CI) is then:

𝐶

𝐼

=

𝜆

max

⁡

−

𝑛

𝑛

−

1

,

CI=

n−1

λ

max

​

−n

​

,

where $n$ is the matrix size (here $n=3$). For our case, $CI = (;3.006 - 3;)/(3-1) = 0.006/2 = 0.003$.

The Random Consistency Index (RI) for $n=3$ is $0.58$ (a known average CI for a randomly filled 3×3 matrix).

Thus, the Consistency Ratio is:

𝐶

𝑅

=

𝐶

𝐼

𝑅

𝐼

≈

0.003

0.58

≈

0.005.

CR=

RI

CI

​

≈

0.58

0.003

​

≈0.005.

We have $CR \approx 0.5%$, which is well below the 10% threshold typically required. This means the judgments are highly consistent. We can proceed with confidence in the derived weights.

Step 4: Decision Matrix for Freight Forwarders

Next, we evaluate 4 hypothetical freight forwarders on the three criteria. We’ll call them Forwarder A, B, C, and D. We assign realistic performance values for each criterion (based on typical logistics scenarios):

Cost: average shipping cost per standard job (in USD; lower is better).

Reliability: on-time delivery percentage (%, higher is better).

Responsiveness: average response time to inquiries (in hours; lower is better).

Table 3. Performance of four freight forwarders on three criteria

Forwarder Cost (USD) Reliability (% on-time) Responsiveness (hours)

A 1200 92% 5 h

B 1000 85% 9 h

C 1300 97% 3 h

D 1100 90% 4 h

Interpretation: Forwarder B offers the cheapest cost ($1000) but has the lowest reliability (85% on-time) and slowest response. Forwarder C is the most expensive ($1300) but excels in reliability (97%) and responsiveness (3h). Forwarders A and D are in between, with D being slightly cheaper and more responsive than A, while A has a bit higher reliability. This decision matrix (alternatives vs criteria values) will be the input to the TOPSIS method.

Step 5: TOPSIS – Normalization and Ideal Solutions

TOPSIS (Technique for Order Preference by Similarity to Ideal Solution) will rank the forwarders by how close each one is to an ideal best case and how far from a worst case. The steps include normalization of the decision matrix, applying criteria weights, identifying ideal and anti-ideal solutions, and computing distances. 5a. Normalize the decision matrix: We convert all criteria measures to dimensionless units by vector normalization. For each criterion $j$, we compute:

𝑟

𝑖

𝑗

=

𝑥

𝑖

𝑗

∑

𝑘

=

1

𝑚

𝑥

𝑘

𝑗

2

,

r

ij

​

=

∑

k=1

m

​

x

kj

2

​

​

x

ij

​

​

, where $x\_{ij}$ is performance of alternative $i$ on criterion $j$, and $m$ is the number of alternatives. This ensures each column of $r\_{ij}$ has length 1. After normalization, we have (rounded to 3 decimal places):

Cost: $\sqrt{1200^2+1000^2+1300^2+1100^2} \approx 2310.84$. So:

– A: $1200/2310.84 = 0.519$,

– B: $1000/2310.84 = 0.433$,

– C: $1300/2310.84 = 0.562$,

– D: $1100/2310.84 = 0.476$.

Reliability: $\sqrt{92^2+85^2+97^2+90^2} \approx 182.20$. So:

– A: $92/182.20 = 0.505$,

– B: $85/182.20 = 0.467$,

– C: $97/182.20 = 0.532$,

– D: $90/182.20 = 0.494$.

Responsiveness: $\sqrt{5^2+9^2+3^2+4^2} \approx 11.446$. So:

– A: $5/11.446 = 0.437$,

– B: $9/11.446 = 0.787$,

– C: $3/11.446 = 0.262$,

– D: $4/11.446 = 0.350$.

5b. Apply criteria weights: We multiply each normalized value by the corresponding criterion weight from Step 2 to get the weighted normalized matrix $v\_{ij} = w\_j \cdot r\_{ij}$. This incorporates the relative importance of each criterion into the scores. The weighted values (using $w\_{\text{cost}}=0.329$, $w\_{\text{rel}}=0.434$, $w\_{\text{resp}}=0.237$) are:

Forwarder A:

– Cost: $0.519 \times 0.329 = 0.171$

– Reliability: $0.505 \times 0.434 = 0.219$

– Responsiveness: $0.437 \times 0.237 = 0.104$ (Total weighted score components for A: [0.171, 0.219, 0.104])

Forwarder B:

– Cost: $0.433 \times 0.329 = 0.142$

– Reliability: $0.467 \times 0.434 = 0.203$

– Responsiveness: $0.787 \times 0.237 = 0.187$ (Components for B: [0.142, 0.203, 0.187])

Forwarder C:

– Cost: $0.562 \times 0.329 = 0.185$

– Reliability: $0.532 \times 0.434 = 0.231$

– Responsiveness: $0.262 \times 0.237 = 0.062$ (Components for C: [0.185, 0.231, 0.062])

Forwarder D:

– Cost: $0.476 \times 0.329 = 0.157$

– Reliability: $0.494 \times 0.434 = 0.214$

– Responsiveness: $0.350 \times 0.237 = 0.083$ (Components for D: [0.157, 0.214, 0.083])

5c. Determine the ideal and anti-ideal solutions: Based on the weighted matrix, we identify the ideal best ($v^+$) and ideal worst ($v^-$) values for each criterion:

For Cost (a cost-type criterion where lower is better):

– $v\_{\text{cost}}^{+} = \min{0.171, 0.142, 0.185, 0.157} = 0.142$ (the smallest weighted cost, from Forwarder B).

– $v\_{\text{cost}}^{-} = \max{0.171, 0.142, 0.185, 0.157} = 0.185$ (the highest cost, from Forwarder C).

For Reliability (a benefit criterion, higher is better):

– $v\_{\text{rel}}^{+} = \max{0.219, 0.203, 0.231, 0.214} = 0.231$ (the highest weighted reliability, from Forwarder C).

– $v\_{\text{rel}}^{-} = \min{0.219, 0.203, 0.231, 0.214} = 0.203$ (the lowest reliability, from Forwarder B).

For Responsiveness (cost-type, lower is better since it’s response time):

– $v\_{\text{resp}}^{+} = \min{0.104, 0.187, 0.062, 0.083} = 0.062$ (fastest response, from Forwarder C).

– $v\_{\text{resp}}^{-} = \max{0.104, 0.187, 0.062, 0.083} = 0.187$ (slowest response, from Forwarder B).

So the ideal solution is

𝑣

+

=

(

0.142

,

0.231

,

0.062

)

v

+

=(0.142,0.231,0.062), which represents a hypothetical forwarder with B’s low cost, C’s high reliability, and C’s quick responsiveness – essentially the best of each criterion. The anti-ideal (worst) solution is

𝑣

−

=

(

0.185

,

0.203

,

0.187

)

v

−

=(0.185,0.203,0.187), combining C’s high cost, B’s poor reliability, and B’s slow response – the worst of each criterion.

Step 6: Closeness Coefficients and Ranking Results

Now we calculate each forwarder’s Euclidean distance to the ideal and to the worst, and from these derive the closeness coefficient:

Distance to ideal for alternative $i$:

𝑑

𝑖

+

=

∑

𝑗

=

1

𝑛

(

𝑣

𝑖

𝑗

−

𝑣

𝑗

+

)

2

,

d

i

+

​

=

∑

j=1

n

​

(v

ij

​

−v

j

+

​

)

2

​

,

Distance to worst for alternative $i$:

𝑑

𝑖

−

=

∑

𝑗

=

1

𝑛

(

𝑣

𝑖

𝑗

−

𝑣

𝑗

−

)

2

.

d

i

−

​

=

∑

j=1

n

​

(v

ij

​

−v

j

−

​

)

2

​

.

Using the weighted values from step 5b and the ideal points above, we compute these distances for each forwarder, and then the closeness coefficient:

𝐶

𝑖

=

𝑑

𝑖

−

𝑑

𝑖

+

+

𝑑

𝑖

−

,

C

i

​

=

d

i

+

​

+d

i

−

​

d

i

−

​

​

, which lies in [0,1] with a higher value indicating a more preferable option (closer to ideal and farther from worst). Table 4. Distances to ideal/worst and closeness coefficients

Forwarder $d\_i^+$ (to ideal) $d\_i^-$ (to worst) Closeness $C\_i$

A 0.0516 0.0858 0.6243

B 0.1277 0.0424 0.2493

C 0.0424 0.1277 0.7507

D 0.0303 0.1082 0.7812

The closeness score $C\_i$ essentially indicates each forwarder’s relative proximity to the ideal solution (1.0 would be the ideal itself). Based on $C\_i$, we rank the forwarders from highest to lowest: Rank 1: Forwarder D – Closeness 0.7812.

Rank 2: Forwarder C – Closeness 0.7507.

Rank 3: Forwarder A – Closeness 0.6243.

Rank 4: Forwarder B – Closeness 0.2493. Result Interpretation: Forwarder D emerges as the top choice. Even though D doesn’t have the single best score in any criterion, it offers a balanced performance (second-best cost and responsiveness, decent reliability) that places it closest to the ideal compromise. Forwarder C is a close second – it excels in reliability and responsiveness (the best in both), but its high cost drags its overall score down slightly. Forwarder A is third, with moderate cost and good reliability, but slower response time. Finally, B ranks lowest: despite being cheapest, B’s poor reliability and responsiveness make it far from the ideal (reflected in its low closeness value). This shows how DeepCAL’s engine, through TOPSIS, naturally balances the trade-offs: a moderately priced, reliable, responsive forwarder (like D) can outrank an extremely reliable but expensive one (C) or a cheap but unreliable one (B).

Sensitivity Analysis: Impact of Changes in Cost or Reliability

To test the robustness of the decision, we simulate how the ranking would change if a forwarder’s performance improves or worsens:

Scenario 1: Forwarder C lowers its cost. Suppose Forwarder C manages to reduce its cost from $1300 down to $1100 (matching D’s cost) while keeping reliability and responsiveness the same. We recompute TOPSIS with this new value. The criteria weights remain the same, but C’s cost improvement significantly boosts its closeness score to $C\_C \approx 0.896 (from 0.7507). In fact, Forwarder C would become the top-ranked option, overtaking D. The new ranking order would be: C (now best), then D, A, and B last. This underscores that cost can be a decisive factor – lowering cost gave C a substantial advantage since it was already strong in the other criteria.

Scenario 2: Forwarder D’s reliability dips. Now suppose Forwarder D suffers a service issue and its on-time reliability falls from 90% to 80% (with other values unchanged). Re-running the calculations, D’s closeness score drops to about $C\_D \approx 0.69 (from 0.7812). Forwarder C would rise to Rank 1 with $C\_C \approx 0.754$, and D would fall to Rank 2. (The order would be C, D, A, B.) This shows the importance of reliability: a significant dip can hurt a forwarder’s standing. In this scenario, C’s superior reliability allows it to overtake D, even though C still has a cost disadvantage.

Overall, these sensitivity tests highlight that DeepCAL’s recommendations are sensitive to changes in key criteria. A forwarder can improve its ranking by addressing its weaker points (e.g. cutting costs or improving reliability), while a decline in performance on the most weighted criterion can notably reduce its competitiveness. The neutrosophic AHP + TOPSIS framework can easily accommodate updated inputs and re-rank alternatives, making it a useful decision engine for dynamic logistics scenarios.

I can help you animate polylines from Kenya to various destinations using Mapbox. Based on the data you've provided, I'll show you how to create an animated line visualization using Mapbox GL JS.

Here's a code example that demonstrates how to animate a polyline from an origin to multiple destinations:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Animate lines from Kenya to destinations</title>

<meta name="viewport" content="initial-scale=1,maximum-scale=1,user-scalable=no">

<link href="https://api.mapbox.com/mapbox-gl-js/v3.10.0/mapbox-gl.css" rel="stylesheet">

<script src="https://api.mapbox.com/mapbox-gl-js/v3.10.0/mapbox-gl.js"></script>

<script src="https://unpkg.com/@turf/turf@6/turf.min.js"></script>

<style>

body { margin: 0; padding: 0; }

#map { position: absolute; top: 0; bottom: 0; width: 100%; }

.overlay {

position: absolute;

top: 10px;

left: 10px;

background: rgba(255, 255, 255, 0.8);

padding: 10px;

border-radius: 3px;

}

</style>

</head>

<body>

<div id="map"></div>

<div class="overlay">

<button id="animate">Animate Routes</button>

</div>

<script>

// TO MAKE THE MAP APPEAR YOU MUST ADD YOUR ACCESS TOKEN FROM https://account.mapbox.com

mapboxgl.accessToken = 'YOUR\_MAPBOX\_ACCESS\_TOKEN';

const map = new mapboxgl.Map({

container: 'map',

style: 'mapbox://styles/mapbox/dark-v11',

center: [25, 5], // Centered on Africa

zoom: 3

});

// Kenya origin coordinates

const origin = [36.990054, 1.2404475];

// Parse your data to create destinations array

const destinations = [

{ country: 'Zimbabwe', coordinates: [31.08848075, -17.80269125] },

{ country: 'Zambia', coordinates: [28.3174378, 15.4136414] },

{ country: 'Madagascar', coordinates: [47.50866443, -14.71204234] },

{ country: 'Comoros', coordinates: [43.2413774, 11.7209701] },

{ country: 'South Sudan', coordinates: [29.69490516, 7.86237248] },

{ country: 'Ethiopia', coordinates: [38.7675998, 9.0146129] },

{ country: 'Uganda', coordinates: [32.3032312, 1.36815764] },

{ country: 'Rwanda', coordinates: [29.88043595, -1.94365675] },

{ country: 'Burundi', coordinates: [29.3731839, -3.3806734] },

{ country: 'Tanzania', coordinates: [39.2968217, 6.8122141] },

// Add more destinations as needed

];

// Create a GeoJSON source with empty features

const routeSource = {

'type': 'FeatureCollection',

'features': []

};

// Steps for animation

const steps = 100;

map.on('load', () => {

// Add a source for the routes

map.addSource('routes', {

'type': 'geojson',

'data': routeSource

});

// Add a line layer for the routes

map.addLayer({

'id': 'route-lines',

'type': 'line',

'source': 'routes',

'layout': {

'line-cap': 'round',

'line-join': 'round'

},

'paint': {

'line-color': [

'match',

['get', 'country'],

'Zimbabwe', '#ff0000',

'Zambia', '#00ff00',

'Madagascar', '#0000ff',

'Comoros', '#ffff00',

'South Sudan', '#ff00ff',

'Ethiopia', '#00ffff',

'Uganda', '#ff8800',

'Rwanda', '#8800ff',

'Burundi', '#00ff88',

'Tanzania', '#ff0088',

'#ffffff' // default color

],

'line-width': 3,

'line-opacity': 0.8

}

});

// Add click event to the animate button

document.getElementById('animate').addEventListener('click', () => {

animateRoutes();

});

});

function animateRoutes() {

// Clear existing routes

routeSource.features = [];

map.getSource('routes').setData(routeSource);

// Animate each route sequentially

destinations.forEach((destination, index) => {

setTimeout(() => {

animateSingleRoute(destination);

}, index \* 1000); // Start each animation with a delay

});

}

function animateSingleRoute(destination) {

// Create a line from origin to destination

const route = {

'type': 'Feature',

'properties': {

'country': destination.country

},

'geometry': {

'type': 'LineString',

'coordinates': [origin, destination.coordinates]

}

};

// Calculate the distance

const lineDistance = turf.length(route);

const arc = [];

// Draw an arc between the origin & destination

for (let i = 0; i < lineDistance; i += lineDistance / steps) {

const segment = turf.along(route, i);

arc.push(segment.geometry.coordinates);

}

// Update the route with calculated arc coordinates

route.geometry.coordinates = arc;

// Animation counter

let counter = 0;

function animate() {

if (counter < steps) {

// Update the route to the current point

const currentRoute = {

'type': 'Feature',

'properties': {

'country': destination.country

},

'geometry': {

'type': 'LineString',

'coordinates': route.geometry.coordinates.slice(0, counter + 1)

}

};

// Add the current route to the source

routeSource.features = routeSource.features.filter(f =>

f.properties.country !== destination.country

);

routeSource.features.push(currentRoute);

map.getSource('routes').setData(routeSource);

counter++;

requestAnimationFrame(animate);

}

}

animate();

}

</script>

</body>

</html>

This code creates a map centered on Africa and animates polylines from Kenya to various destinations. The animation works by:

1. Setting up a Mapbox GL JS map with a dark style
2. Defining the Kenya origin point and destination countries
3. Creating a GeoJSON source for the routes
4. Adding a line layer with different colors for each country
5. Implementing an animation function that draws each route progressively

The example is based on techniques shown in the [**Animate a line**](https://docs.mapbox.com/mapbox-gl-js/example/animate-a-line/) and [**Animate a point along a route**](https://docs.mapbox.com/mapbox-gl-js/example/animate-point-along-route/) examples from Mapbox.

To use this code:

1. Replace 'YOUR\_MAPBOX\_ACCESS\_TOKEN' with your actual Mapbox access token
2. Customize the colors, animation speed, or add more destinations as needed
3. You can adjust the steps variable to make the animation smoother or faster

The code uses the Turf.js library to calculate points along each route for smooth animation.

Follow these accurately

DeepCAL: Intelligent Logistics and Analytics Dashboard Table of Contents Introduction

System Architecture

UI Pages & Component Breakdown

3.1 Mapview (Main Page)

3.2 Analytics (Shipment, Destination, Freight, Deep Insights)

3.3 DeepCAL Calculator

3.4 Settings

Data Integration & APIs

Database Schema

Security & Performance Considerations

Conclusion

1. Introduction DeepCAL is an AI-powered logistics and analytics platform optimized for emergency response and supply chain efficiency. The system integrates real-time shipment tracking, freight forwarder evaluation, and deep analytical insights, leveraging the Neutrosophic AHP-TOPSIS model for intelligent decision-making.

This documentation outlines the interface structure, components, and data flow of DeepCAL, ensuring a well-defined, scalable, and intuitive user experience.

1. System Architecture DeepCAL follows a modular design, consisting of:

Frontend (Next.js, Tailwind CSS, shadcn/ui) for an interactive user interface.

Backend (FastAPI, Python-based decision engine) for complex analytics and AI computations.

Database (Supabase - PostgreSQL) for data storage, real-time updates, and API services.

Edge Functions (Supabase serverless logic) for background processing and automation.

Each module interacts seamlessly through well-defined REST APIs.

1. UI Pages & Component Breakdown 3.1 Mapview (Main Page) Description A full-screen interactive map for shipment tracking, geospatial risk assessment, and logistics optimization.

Key Features Live Shipment Tracking: Displays real-time shipment locations using Mapbox GL.

Geospatial Risk Indicators: Color-coded overlays for high-risk areas based on real-time data.

Interactive Filtering: Users can toggle shipment categories (e.g., high-priority, delayed, at-risk).

Route Optimization: AI-powered recommendations for alternate routing based on risk analytics.

Component Breakdown Component Functionality Map Widget Displays real-time locations, routes, and heatmaps. Risk Heatmap Highlights high-risk zones for shipments. Filter Controls Allow users to refine displayed shipments by priority, status, etc. Detailed Shipment View Clickable markers provide shipment details, including risk scores. 3.2 Analytics Page Description A dashboard containing multiple analytic insights into shipment performance, risk factors, and freight efficiency.

Tabs & Insights Tab Features Shipment Analytics Graphs and tables on delivery times, cost efficiency, and real-time shipment data. Destination Analytics Evaluates risk levels and delivery success rates across destinations. Freight Analytics Performance and reliability scores of freight forwarders. Deep Insights AI-driven predictions on shipment risks, cost variations, and supply chain bottlenecks. Component Breakdown Component Functionality Bar & Line Charts Displays historical and predictive shipment trends. Performance Scorecards Summarizes key logistics metrics (OTIF %, delay rates, cost deviation). Interactive Data Tables Allows filtering and deep dives into logistics performance. Heatmap View Geospatial visualization of high-risk destinations. 3.3 DeepCAL Calculator Description A real-time logistics decision engine that computes the best freight options, shipment routes, and risk adjustments.

Key Features Neutrosophic AHP-TOPSIS Computation: Uses AI-based ranking models to evaluate shipment alternatives.

Automated Risk Assessment: Dynamically adjusts weightings based on historical and real-time factors.

Cost-Performance Optimization: Recommends the most efficient routes and freight forwarders.

Component Breakdown Component Functionality Decision Engine UI Allows users to input shipment criteria and run calculations. Dynamic Weight Sliders Adjusts factor importance (cost, reliability, responsiveness). Ranking Table Displays optimal options ranked based on AHP-TOPSIS. AI Insights Provides reasoning and recommendations for selected shipment plans. 3.4 Settings Description A configuration panel for managing DeepCAL system preferences, user roles, API integrations, and visual themes.

Key Features General System Settings: Manage analytics update frequency, alert thresholds, and API integrations.

User Role Management: Define access control levels for different stakeholders.

Customization & Themes: Adapt UI themes (light/dark mode), adjust dashboard layout.

API & Integration Management: Configure connections with Supabase, Mapbox, OpenWeather API, etc.

1. Data Integration & APIs DeepCAL integrates with multiple external services and APIs for real-time insights.

API Function Supabase Database API Handles storage of shipment, freight, and risk data. Mapbox API Provides real-time mapping, geospatial risk visualization. OpenWeather API Supplies weather conditions for risk assessment. DeepCAL Decision Engine (FastAPI) Runs AHP-TOPSIS ranking and risk analysis. 5. Database Schema DeepCAL utilizes a normalized PostgreSQL schema in Supabase.

Shipment Table sql Copy Edit CREATE TABLE shipments ( id SERIAL PRIMARY KEY, origin VARCHAR(255), destination VARCHAR(255), weight FLOAT, volume FLOAT, cost FLOAT, status VARCHAR(50), estimated\_arrival TIMESTAMP ); Freight Forwarders Table sql Copy Edit CREATE TABLE freight\_forwarders ( id SERIAL PRIMARY KEY, name VARCHAR(255), service\_type VARCHAR(255), cost\_metric FLOAT, reliability\_score FLOAT ); Risk Assessment Table sql Copy Edit CREATE TABLE risk\_assessment ( id SERIAL PRIMARY KEY, shipment\_id INT REFERENCES shipments(id), risk\_score FLOAT, delay\_probability FLOAT, reason TEXT ); API Request Logging sql Copy Edit CREATE TABLE api\_logs ( id SERIAL PRIMARY KEY, endpoint VARCHAR(255), request\_time TIMESTAMP, response\_time FLOAT, status\_code INT ); 6. Security & Performance Considerations Authentication: Uses Supabase Auth (JWT-based).

Rate Limiting: Prevents excessive API calls.

Caching Strategies: Edge functions optimize data retrieval to improve responsiveness.

Monitoring & Logging: Logs API calls and performance metrics for system health tracking.

1. Conclusion DeepCAL is an advanced AI-powered logistics analytics and decision-making tool, integrating real-time tracking, predictive risk assessment, and automated freight optimization. Its architecture supports scalability, security, and seamless integration with modern APIs and decision-making models.

This document serves as the foundation for accurate implementation and development, ensuring that DeepCAL provides an intuitive and data-driven interface for efficient emergency logistics and supply chain management.

Thought for 3 seconds

**DeepCAL: Technical Documentation Report**

**Table of Contents**

1. [Introduction](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#1-introduction)
2. [System Architecture](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#2-system-architecture)
3. [UI Pages & Component Breakdown](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#3-ui-pages--component-breakdown)
   * [3.1 Dashboard (Main Page)](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#31-dashboard-main-page)
   * [3.2 Live Tracking](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#32-live-tracking)
   * [3.3 Shipments Management](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#33-shipments-management)
   * [3.4 Forwarders Management](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#34-forwarders-management)
   * [3.5 Analytics](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#35-analytics)
4. [Data Integration & APIs](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#4-data-integration--apis)
5. [Database Schema](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#5-database-schema)
6. [Security & Performance Considerations](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#6-security--performance-considerations)
7. [Styling and UI/UX](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#7-styling-and-uiux)
8. [Known Issues and Future Development](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#8-known-issues-and-future-development)
9. [Testing and Deployment](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#9-testing-and-deployment)
10. [Conclusion](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#10-conclusion)

**1. Introduction**

DeepCAL (Deep Crisis Analytics and Logistics) is an intelligent emergency logistics platform designed to optimize the allocation and delivery of aid and resources during crises. The system serves as a comprehensive decision-support tool for emergency response coordinators, logistics managers, and field operators who need to make rapid, data-driven decisions in high-pressure situations.

**Purpose**

DeepCAL's primary purpose is to:

* Provide real-time visibility into shipment locations and statuses
* Assess and monitor risk levels for active shipments
* Evaluate and compare logistics partner performance
* Offer data-driven insights through comprehensive analytics
* Support decision-making through the Neutrosophic AHP-TOPSIS model

**Target Users**

* **Emergency Response Coordinators**: Professionals who oversee the overall emergency response effort and need high-level insights
* **Logistics Managers**: Specialists who manage the movement of resources and need detailed tracking and performance data
* **Field Operators**: Personnel on the ground who need real-time updates on shipment statuses
* **Decision Makers**: Stakeholders who rely on data-driven insights to allocate resources effectively

**Key Features**

* Real-time shipment tracking with geospatial visualization
* Risk assessment and monitoring for all shipments
* Performance evaluation of logistics partners
* Comprehensive analytics dashboard
* Detailed shipment management system
* Responsive design for both desktop and mobile use

**2. System Architecture**

DeepCAL follows a modern, scalable architecture built on Next.js and Supabase, with a clear separation of concerns between frontend, backend, and database components.

**Technology Stack**

**Frontend**

* **Next.js 13+ (App Router)**: React framework for server and client components
* **TypeScript**: For type-safe code development
* **Tailwind CSS**: For utility-first styling
* **shadcn/ui**: For accessible, customizable UI components
* **Recharts**: For interactive data visualization

**Backend**

* **Next.js API Routes**: For server-side logic
* **Supabase Edge Functions**: For background processing and serverless functions
* **Neutrosophic AHP-TOPSIS Model**: Core decision-making algorithm

**Database**

* **Supabase (PostgreSQL)**: For data storage and real-time subscriptions
* **Supabase Auth**: For authentication and authorization

**Architecture Diagram**

DeepCAL Architecture

Client BrowserNext.js FrontendNext.js API RoutesSupabase ClientSupabase BackendPostgreSQL DatabaseSupabase AuthSupabase RealtimeNeutrosophic AHP-TOPSIS Model

**Component Interaction**

1. **Client-Side Rendering**:
   * React components render UI elements
   * Client-side state management handles user interactions
   * Supabase client fetches data and subscribes to real-time updates
2. **Server-Side Rendering**:
   * Next.js server components fetch data directly from Supabase
   * Pre-rendered HTML is sent to the client for faster initial load
   * Hydration connects the pre-rendered HTML with React's interactivity
3. **Data Flow**:
   * Supabase provides real-time data through subscriptions
   * API routes handle complex operations and business logic
   * Database triggers automate certain workflows

**3. UI Pages & Component Breakdown**

**3.1 Dashboard (Main Page)**

**Description**

The dashboard serves as the application's home page, providing a comprehensive overview of emergency logistics operations with key metrics, charts, and recent shipment information.

**Key Features**

* **Key Performance Indicators**: Display of critical metrics like total shipments, active shipments, high-risk shipments, and delayed shipments
* **Risk Distribution Chart**: Visualization of shipments by risk level over time
* **Forwarder Performance**: Comparison of logistics partner performance metrics
* **Recent Shipments**: Table of the latest shipment activities with status indicators

**Component Breakdown**

| **Component** | **File Path** | **Functionality** |
| --- | --- | --- |
| Dashboard Page | app/page.tsx | Main container with Suspense boundaries |
| Dashboard Header | components/dashboard/dashboard-header.tsx | Title, description, and filter controls |
| Dashboard Stats | components/dashboard/dashboard-stats.tsx | Key metrics with trend indicators |
| Shipment Risk Chart | components/dashboard/shipment-risk-chart.tsx | Area chart of risk distribution |
| Forwarder Performance | components/dashboard/forwarder-performance.tsx | Bar chart of forwarder metrics |
| Recent Shipments | components/dashboard/recent-shipments.tsx | Table of recent shipment activities |

**Code Example: Dashboard Stats Component**

export function DashboardStats({ stats }: DashboardStatsProps) {

const statCards = [

{

title: "Total Shipments",

value: stats.total\_shipments,

icon: Box,

color: "text-blue-500",

bgColor: "bg-blue-500/10",

},

{

title: "Active Shipments",

value: stats.active\_shipments,

icon: Truck,

color: "text-green-500",

bgColor: "bg-green-500/10",

},

{

title: "High Risk",

value: stats.high\_risk\_shipments,

icon: AlertTriangle,

color: "text-red-500",

bgColor: "bg-red-500/10",

},

{

title: "Delayed",

value: stats.delayed\_shipments,

icon: Clock,

color: "text-amber-500",

bgColor: "bg-amber-500/10",

},

]

return (

<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">

{statCards.map((card) => (

<Card key={card.title} className="overflow-hidden">

<div

className="absolute top-0 right-0 h-16 w-16 opacity-10 rounded-bl-full bg-gradient-to-br from-transparent to-current"

style={{ color: card.color.replace("text-", "") }}

/>

<CardHeader className="flex flex-row items-center justify-between pb-2">

<CardTitle className="text-sm font-medium">{card.title}</CardTitle>

<div className={`${card.bgColor} ${card.color} p-2 rounded-full`}>

<card.icon className="h-4 w-4" />

</div>

</CardHeader>

<CardContent>

<div className="text-2xl font-bold">{card.value.toLocaleString()}</div>

<div className="text-xs text-muted-foreground mt-1">

{Math.random() > 0.5 ? (

<span className="text-green-500">↑ {Math.floor(Math.random() \* 10) + 1}%</span>

) : (

<span className="text-red-500">↓ {Math.floor(Math.random() \* 10) + 1}%</span>

)}{" "}

from last period

</div>

</CardContent>

</Card>

))}

</div>

)

}

**3.2 Live Tracking**

**Description**

The Live Tracking page provides real-time monitoring of active shipments with an interactive map, a list of active shipments, and a detailed sidebar for selected shipments.

**Key Features**

* **Interactive Map**: Displays shipment locations with risk-based color coding
* **Active Shipment List**: Shows currently active shipments with key information
* **Shipment Details Sidebar**: Provides comprehensive information about selected shipments
* **Real-time Updates**: Automatically updates when shipment data changes

**Component Breakdown**

| **Component** | **File Path** | **Functionality** |
| --- | --- | --- |
| Live Tracking Page | app/live-tracking/page.tsx | Main container with layout structure |
| Live Tracking Header | components/live-tracking/live-tracking-header.tsx | Title, search, and filter controls |
| Live Tracking Map | components/live-tracking/live-tracking-map.tsx | Interactive map with shipment markers |
| Live Shipment List | components/live-tracking/live-shipment-list.tsx | Real-time list of active shipments |
| Shipment Details Sidebar | components/live-tracking/shipment-details-sidebar.tsx | Slide-in panel with detailed information |

**Code Example: Real-time Subscription**

*// Subscribe to real-time updates*

const subscription = supabase

.channel("shipment-location-changes")

.on(

"postgres\_changes",

{

event: "UPDATE",

schema: "public",

table: "shipments",

filter: "location=neq.null",

},

(payload) => {

const updatedShipment = payload.new as Shipment

setShipments((current) => {

const index = current.findIndex((s) => s.id === updatedShipment.id)

if (index >= 0) {

const updated = [...current]

updated[index] = updatedShipment

return updated

} else {

return [...current, updatedShipment]

}

})

},

)

.subscribe()

*// Clean up subscription on component unmount*

return () => {

subscription.unsubscribe()

}

**3.3 Shipments Management**

**Description**

The Shipments Management section provides comprehensive tools for viewing, filtering, and managing all shipments in the system, with detailed views for individual shipments.

**Key Features**

* **Shipments Table**: Interactive table with sorting, filtering, and pagination
* **Shipment Details**: Comprehensive view of a specific shipment
* **Shipment Timeline**: Chronological view of shipment events
* **Shipment Map**: Location tracking for individual shipments
* **Shipment Actions**: Interface for taking actions on shipments

**Component Breakdown**

| **Component** | **File Path** | **Functionality** |
| --- | --- | --- |
| Shipments Page | app/shipments/page.tsx | Main shipments listing page |
| Shipments Header | components/shipments/shipments-header.tsx | Title, search, and filter controls |
| Shipments Table | components/shipments/shipments-table.tsx | Server component for data fetching |
| Shipments Data Table | components/shipments/shipments-data-table.tsx | Interactive client-side table |
| Shipment Details Page | app/shipments/[id]/page.tsx | Individual shipment view |
| Shipment Header | components/shipments/shipment-header.tsx | Shipment title and status indicators |
| Shipment Details | components/shipments/shipment-details.tsx | Detailed shipment information |
| Shipment Timeline | components/shipments/shipment-timeline.tsx | Chronological event display |
| Shipment Map | components/shipments/shipment-map.tsx | Location tracking map |
| Shipment Actions | components/shipments/shipment-actions.tsx | Action buttons for shipment management |

**Code Example: Shipments Data Table**

export function ShipmentsDataTable({ data }: ShipmentsDataTableProps) {

const [sorting, setSorting] = useState<SortingState>([])

const [columnFilters, setColumnFilters] = useState<ColumnFiltersState>([])

const [columnVisibility, setColumnVisibility] = useState<VisibilityState>({})

const [rowSelection, setRowSelection] = useState({})

const table = useReactTable({

data,

columns,

onSortingChange: setSorting,

onColumnFiltersChange: setColumnFilters,

getCoreRowModel: getCoreRowModel(),

getPaginationRowModel: getPaginationRowModel(),

getSortedRowModel: getSortedRowModel(),

getFilteredRowModel: getFilteredRowModel(),

onColumnVisibilityChange: setColumnVisibility,

onRowSelectionChange: setRowSelection,

state: {

sorting,

columnFilters,

columnVisibility,

rowSelection,

},

})

return (

<div className="w-full">

{*/\* Table controls \*/*}

<div className="flex items-center py-4">

<DropdownMenu>

<DropdownMenuTrigger asChild>

<Button variant="outline" className="ml-auto">

Columns <ChevronDown className="ml-2 h-4 w-4" />

</Button>

</DropdownMenuTrigger>

<DropdownMenuContent align="end">

{table

.getAllColumns()

.filter((column) => column.getCanHide())

.map((column) => {

return (

<DropdownMenuCheckboxItem

key={column.id}

className="capitalize"

checked={column.getIsVisible()}

onCheckedChange={(value) => column.toggleVisibility(!!value)}

>

{column.id}

</DropdownMenuCheckboxItem>

)

})}

</DropdownMenuContent>

</DropdownMenu>

</div>

{*/\* Table \*/*}

<div className="rounded-md border">

<Table>

<TableHeader>

{table.getHeaderGroups().map((headerGroup) => (

<TableRow key={headerGroup.id}>

{headerGroup.headers.map((header) => {

return (

<TableHead key={header.id}>

{header.isPlaceholder ? null : flexRender(header.column.columnDef.header, header.getContext())}

</TableHead>

)

})}

</TableRow>

))}

</TableHeader>

<TableBody>

{table.getRowModel().rows?.length ? (

table.getRowModel().rows.map((row) => (

<TableRow

key={row.id}

data-state={row.getIsSelected() && "selected"}

className="group hover:bg-muted/50"

>

{row.getVisibleCells().map((cell) => (

<TableCell key={cell.id}>{flexRender(cell.column.columnDef.cell, cell.getContext())}</TableCell>

))}

</TableRow>

))

) : (

<TableRow>

<TableCell colSpan={columns.length} className="h-24 text-center">

No results.

</TableCell>

</TableRow>

)}

</TableBody>

</Table>

</div>

{*/\* Pagination \*/*}

<div className="flex items-center justify-end space-x-2 py-4">

<div className="flex-1 text-sm text-muted-foreground">

{table.getFilteredSelectedRowModel().rows.length} of {table.getFilteredRowModel().rows.length} row(s)

selected.

</div>

<div className="space-x-2">

<Button

variant="outline"

size="sm"

onClick={() => table.previousPage()}

disabled={!table.getCanPreviousPage()}

>

Previous

</Button>

<Button variant="outline" size="sm" onClick={() => table.nextPage()} disabled={!table.getCanNextPage()}>

Next

</Button>

</div>

</div>

</div>

)

}

**3.4 Forwarders Management**

**Description**

The Forwarders Management section provides tools for viewing and evaluating logistics partners, with performance metrics and analytics.

**Key Features**

* **Forwarders Grid**: Card-based display of logistics partners
* **Performance Metrics**: Visual indicators of forwarder performance
* **Quick Actions**: Interface for managing forwarders

**Component Breakdown**

| **Component** | **File Path** | **Functionality** |
| --- | --- | --- |
| Forwarders Page | app/forwarders/page.tsx | Main forwarders listing page |
| Forwarders Header | components/forwarders/forwarders-header.tsx | Title, search, and filter controls |
| Forwarders Grid | components/forwarders/forwarders-grid.tsx | Server component for data fetching |
| Forwarder Card | components/forwarders/forwarder-card.tsx | Card display for individual forwarders |

**Code Example: Forwarder Card**

export function ForwarderCard({ forwarder }: ForwarderCardProps) {

const getScoreColor = (score: number) => {

if (score >= 8) return "text-green-500"

if (score >= 6) return "text-amber-500"

return "text-red-500"

}

return (

<Card className="overflow-hidden transition-all hover:shadow-md">

<div

className="absolute top-0 right-0 h-24 w-24 opacity-5 rounded-bl-full bg-gradient-to-br from-transparent to-current"

style={{ color: getScoreColor(forwarder.performance\_score).replace("text-", "") }}

/>

<CardHeader className="flex flex-row items-start justify-between pb-2">

<div className="space-y-1">

<h3 className="font-semibold text-lg leading-none">{forwarder.name}</h3>

<p className="text-sm text-muted-foreground">Logistics Partner</p>

</div>

<DropdownMenu>

<DropdownMenuTrigger asChild>

<Button variant="ghost" size="icon" className="h-8 w-8">

<MoreHorizontal className="h-4 w-4" />

</Button>

</DropdownMenuTrigger>

<DropdownMenuContent align="end">

<DropdownMenuLabel>Actions</DropdownMenuLabel>

<DropdownMenuItem>View details</DropdownMenuItem>

<DropdownMenuItem>Edit forwarder</DropdownMenuItem>

<DropdownMenuSeparator />

<DropdownMenuItem>Contact forwarder</DropdownMenuItem>

<DropdownMenuItem className="text-red-600">Remove forwarder</DropdownMenuItem>

</DropdownMenuContent>

</DropdownMenu>

</CardHeader>

<CardContent className="pb-0">

<div className="grid grid-cols-2 gap-4">

<div className="flex flex-col items-center justify-center p-3 bg-muted/50 rounded-lg">

<div className="flex items-center gap-1 text-sm text-muted-foreground mb-1">

<Star className="h-4 w-4" />

<span>Performance</span>

</div>

<div className={`text-2xl font-bold ${getScoreColor(forwarder.performance\_score)}`}>

{forwarder.performance\_score.toFixed(1)}

</div>

<div className="text-xs text-muted-foreground">out of 10</div>

</div>

<div className="flex flex-col items-center justify-center p-3 bg-muted/50 rounded-lg">

<div className="flex items-center gap-1 text-sm text-muted-foreground mb-1">

<Truck className="h-4 w-4" />

<span>On-Time Rate</span>

</div>

<div

className={`text-2xl font-bold ${

forwarder.on\_time\_rate >= 0.9

? "text-green-500"

: forwarder.on\_time\_rate >= 0.7

? "text-amber-500"

: "text-red-500"

}`}

>

{(forwarder.on\_time\_rate \* 100).toFixed(0)}%

</div>

<div className="text-xs text-muted-foreground">of deliveries</div>

</div>

</div>

<div className="mt-4 space-y-2">

<div className="flex items-center justify-between">

<div className="flex items-center gap-2 text-sm text-muted-foreground">

<Package className="h-4 w-4" />

<span>Active Shipments</span>

</div>

<span className="font-medium">{forwarder.active\_shipments}</span>

</div>

<div className="flex items-center justify-between">

<div className="flex items-center gap-2 text-sm text-muted-foreground">

<Package className="h-4 w-4" />

<span>Total Shipments</span>

</div>

<span className="font-medium">{forwarder.total\_shipments}</span>

</div>

</div>

</CardContent>

<CardFooter className="flex justify-between mt-6">

<Button variant="ghost" size="sm" asChild>

<Link href={`/forwarders/${forwarder.id}`}>

<ExternalLink className="h-4 w-4 mr-2" />

Details

</Link>

</Button>

<Button variant="ghost" size="sm" asChild>

<Link href={`/analytics?forwarder=${forwarder.id}`}>

<BarChart className="h-4 w-4 mr-2" />

Analytics

</Link>

</Button>

</CardFooter>

</Card>

)

}

**3.5 Analytics**

**Description**

The Analytics section provides comprehensive data visualization and insights into shipment trends, risk distribution, forwarder performance, and route metrics.

**Key Features**

* **Analytics Overview**: Key metrics and trends
* **Shipment Trends Chart**: Visualization of shipment volumes over time
* **Risk Distribution Chart**: Breakdown of shipments by risk level
* **Forwarder Comparison Chart**: Performance comparison across logistics partners
* **Performance Metrics Table**: Detailed metrics for shipping routes

**Component Breakdown**

| **Component** | **File Path** | **Functionality** |
| --- | --- | --- |
| Analytics Page | app/analytics/page.tsx | Main analytics dashboard |
| Analytics Header | components/analytics/analytics-header.tsx | Title, date range, and filter controls |
| Analytics Overview | components/analytics/analytics-overview.tsx | Key metrics overview |
| Shipment Trends Chart | components/analytics/shipment-trends-chart.tsx | Line chart of shipment trends |
| Risk Distribution Chart | components/analytics/risk-distribution-chart.tsx | Pie chart of risk distribution |
| Forwarder Comparison Chart | components/analytics/forwarder-comparison-chart.tsx | Bar chart comparing forwarders |
| Performance Metrics Table | components/analytics/performance-metrics-table.tsx | Table of route performance metrics |

**Code Example: Risk Distribution Chart**

export function RiskDistributionChart() {

return (

<Card>

<CardHeader>

<CardTitle>Risk Distribution</CardTitle>

<CardDescription>Breakdown of shipments by risk level</CardDescription>

</CardHeader>

<CardContent className="h-[350px]">

<ResponsiveContainer width="100%" height="100%">

<PieChart>

<Pie

data={data}

cx="50%"

cy="50%"

labelLine={false}

outerRadius={120}

innerRadius={60}

fill="#8884d8"

dataKey="value"

label={({ name, percent }) => `${name}: ${(percent \* 100).toFixed(0)}%`}

>

{data.map((entry, index) => (

<Cell key={`cell-${index}`} fill={entry.color} />

))}

</Pie>

<Tooltip

content={({ active, payload }) => {

if (active && payload && payload.length) {

const data = payload[0].payload

return (

<div className="rounded-lg border bg-background p-2 shadow-sm">

<div className="font-bold">{data.name}</div>

<div className="text-sm">

<span className="font-medium">{data.value}</span> shipments

</div>

<div className="text-sm">

<span className="font-medium">{((data.value / 1150) \* 100).toFixed(1)}%</span> of total

</div>

</div>

)

}

return null

}}

/>

<Legend />

</PieChart>

</ResponsiveContainer>

</CardContent>

</Card>

)

}

**4. Data Integration & APIs**

DeepCAL integrates with Supabase for data storage, authentication, and real-time updates. The application uses a combination of server-side and client-side data fetching strategies.

**Supabase Client Initialization**

*// lib/supabase.ts*

import { createClient } from "@supabase/supabase-js"

import type { Database } from "@/lib/database.types"

const supabaseUrl = process.env.NEXT\_PUBLIC\_SUPABASE\_URL!

const supabaseAnonKey = process.env.NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY!

export const supabase = createClient<Database>(supabaseUrl, supabaseAnonKey)

*// Server-side client for use in Server Components and API routes*

export const createServerSupabaseClient = async () => {

const { cookies } = await import("next/headers")

const cookieStore = cookies()

return createClient<Database>(supabaseUrl, supabaseAnonKey, {

cookies: {

get(name: string) {

return cookieStore.get(name)?.value

},

},

})

}

**Data Fetching Strategies**

**Server-Side Data Fetching**

Server components fetch data directly from Supabase using the server client:

*// Example from components/shipments/shipments-table.tsx*

export async function ShipmentsTable() {

const supabase = await createServerSupabaseClient()

*// Fetch shipments with forwarder information*

const { data: shipments, error } = await supabase

.from("shipments")

.select(`

id,

request\_reference,

origin,

destination,

status,

risk\_level,

estimated\_arrival,

actual\_arrival,

priority,

created\_at,

forwarder\_id,

forwarders(name)

`)

.order("created\_at", { ascending: false })

if (error) {

console.error("Error fetching shipments:", error)

throw new Error("Failed to fetch shipments")

}

return <ShipmentsDataTable data={shipments || []} />

}

**Client-Side Data Fetching**

Client components fetch data using the browser client, typically for interactive features:

*// Example from components/live-tracking/live-tracking-map.tsx*

useEffect(() => {

*// Initial fetch of shipments with location data*

const fetchShipments = async () => {

try {

const { data, error } = await supabase

.from("shipments")

.select("\*")

.not("location", "is", null)

.order("risk\_level", { ascending: false })

if (error) throw error

setShipments(data || [])

} catch (error) {

console.error("Error fetching shipments:", error)

toast({

title: "Error fetching shipments",

description: "Could not load shipment location data",

variant: "destructive",

})

} finally {

setIsLoading(false)

}

}

fetchShipments()

*// ...*

}, [toast])

**Real-Time Subscriptions**

DeepCAL leverages Supabase's real-time capabilities for live updates:

*// Subscribe to shipment status changes*

const subscription = supabase

.channel("shipment-status-changes")

.on(

"postgres\_changes",

{

event: "\*",

schema: "public",

table: "shipments",

},

(payload) => {

if (payload.eventType === "UPDATE") {

const updatedShipment = payload.new as Shipment

setShipments((current) => {

const index = current.findIndex((s) => s.id === updatedShipment.id)

if (index >= 0) {

const updated = [...current]

updated[index] = updatedShipment

return updated

} else if (updatedShipment.status === "in transit") {

return [...current, updatedShipment].sort((a, b) => b.risk\_level - a.risk\_level).slice(0, 10)

}

return current

})

} else if (payload.eventType === "INSERT") {

const newShipment = payload.new as Shipment

if (newShipment.status === "in transit") {

setShipments((current) =>

[...current, newShipment].sort((a, b) => b.risk\_level - a.risk\_level).slice(0, 10),

)

}

}

},

)

.subscribe()

**API Endpoints**

While the current implementation primarily uses Supabase directly, the application is structured to support API routes for more complex operations:

**Potential API Routes**

* /api/shipments/calculate-risk: Calculate risk scores for shipments
* /api/analytics/generate-report: Generate comprehensive analytics reports
* /api/forwarders/evaluate: Evaluate forwarder performance based on historical data

**5. Database Schema**

DeepCAL uses a Supabase PostgreSQL database with the following schema:

**Shipments Table**

CREATE TABLE shipments (

id UUID PRIMARY KEY DEFAULT uuid\_generate\_v4(),

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

request\_reference TEXT NOT NULL,

origin TEXT NOT NULL,

destination TEXT NOT NULL,

status TEXT NOT NULL,

risk\_level INTEGER NOT NULL,

estimated\_arrival TIMESTAMP WITH TIME ZONE NOT NULL,

actual\_arrival TIMESTAMP WITH TIME ZONE,

forwarder\_id UUID REFERENCES forwarders(id),

customer\_id UUID REFERENCES customers(id),

priority TEXT CHECK (priority IN ('low', 'medium', 'high', 'critical')),

location JSONB,

CONSTRAINT valid\_location CHECK (location IS NULL OR (location ? 'lat' AND location ? 'lng'))

);

**Forwarders Table**

CREATE TABLE forwarders (

id UUID PRIMARY KEY DEFAULT uuid\_generate\_v4(),

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

name TEXT NOT NULL,

performance\_score NUMERIC NOT NULL,

active\_shipments INTEGER NOT NULL,

total\_shipments INTEGER NOT NULL,

on\_time\_rate NUMERIC NOT NULL,

CONSTRAINT valid\_performance\_score CHECK (performance\_score >= 0 AND performance\_score <= 10),

CONSTRAINT valid\_on\_time\_rate CHECK (on\_time\_rate >= 0 AND on\_time\_rate <= 1)

);

**Users Table**

CREATE TABLE users (

id UUID PRIMARY KEY,

email TEXT UNIQUE NOT NULL,

role TEXT CHECK (role IN ('admin', 'operator', 'viewer')),

name TEXT NOT NULL

);

**TypeScript Type Definitions**

The database schema is represented in TypeScript for type safety:

*// lib/database.types.ts*

export type Json = string | number | boolean | null | { [key: string]: Json | undefined } | Json[]

export interface Database {

public: {

Tables: {

shipments: {

Row: {

id: string

created\_at: string

updated\_at: string

request\_reference: string

origin: string

destination: string

status: string

risk\_level: number

estimated\_arrival: string

actual\_arrival: string | null

forwarder\_id: string

customer\_id: string

priority: "low" | "medium" | "high" | "critical"

location: { lat: number; lng: number } | null

}

Insert: {

id?: string

created\_at?: string

updated\_at?: string

request\_reference: string

origin: string

destination: string

status: string

risk\_level: number

estimated\_arrival: string

actual\_arrival?: string | null

forwarder\_id: string

customer\_id: string

priority: "low" | "medium" | "high" | "critical"

location?: { lat: number; lng: number } | null

}

Update: {

id?: string

created\_at?: string

updated\_at?: string

request\_reference?: string

origin?: string

destination?: string

status?: string

risk\_level?: number

estimated\_arrival?: string

actual\_arrival?: string | null

forwarder\_id?: string

customer\_id?: string

priority?: "low" | "medium" | "high" | "critical"

location?: { lat: number; lng: number } | null

}

}

forwarders: {

Row: {

id: string

created\_at: string

name: string

performance\_score: number

active\_shipments: number

total\_shipments: number

on\_time\_rate: number

}

Insert: {

id?: string

created\_at?: string

name: string

performance\_score: number

active\_shipments: number

total\_shipments: number

on\_time\_rate: number

}

Update: {

id?: string

created\_at?: string

name?: string

performance\_score?: number

active\_shipments?: number

total\_shipments?: number

on\_time\_rate?: number

}

}

users: {

Row: {

id: string

email: string

role: "admin" | "operator" | "viewer"

name: string

}

Insert: {

id: string

email: string

role: "admin" | "operator" | "viewer"

name: string

}

Update: {

id?: string

email?: string

role?: "admin" | "operator" | "viewer"

name?: string

}

}

}

}

}

**6. Security & Performance Considerations**

**Authentication and Authorization**

While not fully implemented in the current codebase, the foundation for authentication and authorization is in place:

* **Supabase Auth**: The application is structured to use Supabase Auth for authentication
* **Role-Based Access Control**: The users table includes a role field for different access levels
* **Server-Side Validation**: Server components can validate user permissions before rendering sensitive data

**Data Security**

* **Type Safety**: TypeScript provides type checking for database operations
* **Input Validation**: Form inputs should be validated before submission
* **Error Handling**: Comprehensive error handling prevents exposure of sensitive information

**Performance Optimization**

**Server-Side Rendering**

Next.js server components improve initial load performance by:

* Fetching data on the server
* Sending pre-rendered HTML to the client
* Reducing client-side JavaScript

*// Example of a server component with Suspense*

export default async function DashboardPage() {

return (

<div className="flex flex-col gap-6 p-6 md:p-8">

<DashboardHeader />

<Suspense fallback={<DashboardSkeleton />}>

<DashboardContent />

</Suspense>

</div>

)

}

**Progressive Loading**

The application uses Suspense and skeleton loaders for a better user experience:

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<ShipmentTrendsChart />

</Suspense>

**Optimized Real-Time Subscriptions**

Real-time subscriptions are optimized with:

* Specific filters to reduce unnecessary updates
* Cleanup on component unmount
* Efficient state updates

**7. Styling and UI/UX**

**Styling Approach**

DeepCAL uses a combination of Tailwind CSS and shadcn/ui components for a consistent, accessible, and customizable UI.

**Tailwind CSS Configuration**

*// tailwind.config.ts*

import type { Config } from "tailwindcss"

const config: Config = {

darkMode: ["class"],

content: [

"./pages/\*\*/\*.{js,ts,jsx,tsx,mdx}",

"./components/\*\*/\*.{js,ts,jsx,tsx,mdx}",

"./app/\*\*/\*.{js,ts,jsx,tsx,mdx}",

"\*.{js,ts,jsx,tsx,mdx}",

],

theme: {

container: {

center: true,

padding: "2rem",

screens: {

"2xl": "1400px",

},

},

extend: {

colors: {

border: "hsl(var(--border))",

input: "hsl(var(--input))",

ring: "hsl(var(--ring))",

background: "hsl(var(--background))",

foreground: "hsl(var(--foreground))",

primary: {

DEFAULT: "hsl(var(--primary))",

foreground: "hsl(var(--primary-foreground))",

},

*// Additional color definitions...*

sidebar: {

DEFAULT: "hsl(var(--sidebar-background))",

foreground: "hsl(var(--sidebar-foreground))",

*// Additional sidebar-specific colors...*

},

},

*// Additional theme extensions...*

},

},

plugins: [require("tailwindcss-animate")],

}

export default config

**CSS Variables**

*/\* globals.css \*/*

@layer base {

:root {

--background: 0 0% 100%;

--foreground: 240 10% 3.9%;

--card: 0 0% 100%;

--card-foreground: 240 10% 3.9%;

--popover: 0 0% 100%;

--popover-foreground: 240 10% 3.9%;

--primary: 240 5.9% 10%;

--primary-foreground: 0 0% 98%;

*/\* Additional light mode variables... \*/*

*/\* Sidebar variables \*/*

--sidebar-background: 0 0% 98%;

--sidebar-foreground: 240 5.3% 26.1%;

*/\* Additional sidebar variables... \*/*

}

.dark {

--background: 240 10% 3.9%;

--foreground: 0 0% 98%;

--card: 240 10% 3.9%;

--card-foreground: 0 0% 98%;

*/\* Additional dark mode variables... \*/*

*/\* Sidebar variables \*/*

--sidebar-background: 240 5.9% 10%;

--sidebar-foreground: 240 4.8% 95.9%;

*/\* Additional dark sidebar variables... \*/*

}

}

**SuperFuturistic UI/UX Design**

The application incorporates several futuristic design elements:

**Animated Logo**

<div className="relative w-8 h-8">

<div className="absolute inset-0 bg-gradient-to-br from-primary to-indigo-600 rounded-lg animate-pulse"></div>

<div className="absolute inset-0.5 bg-background rounded-lg flex items-center justify-center">

<span className="font-bold text-primary">DC</span>

</div>

</div>

**Card Design with Gradient Accents**

<Card key={card.title} className="overflow-hidden">

<div

className="absolute top-0 right-0 h-16 w-16 opacity-10 rounded-bl-full bg-gradient-to-br from-transparent to-current"

style={{ color: card.color.replace("text-", "") }}

/>

<CardHeader className="flex flex-row items-center justify-between pb-2">

<CardTitle className="text-sm font-medium">{card.title}</CardTitle>

<div className={`${card.bgColor} ${card.color} p-2 rounded-full`}>

<card.icon className="h-4 w-4" />

</div>

</CardHeader>

<CardContent>

{*/\* Card content... \*/*}

</CardContent>

</Card>

**Interactive Charts with Custom Tooltips**

<ResponsiveContainer width="100%" height="100%">

<AreaChart

data={riskData}

margin={{ top: 10, right: 10, left: 0, bottom: 0 }}

onMouseMove={(e) => {

if (e.activeTooltipIndex !== undefined) {

setActiveIndex(e.activeTooltipIndex)

}

}}

onMouseLeave={() => setActiveIndex(null)}

>

<defs>

<linearGradient id="colorLow" x1="0" y1="0" x2="0" y2="1">

<stop offset="5%" stopColor="#10b981" stopOpacity={0.8} />

<stop offset="95%" stopColor="#10b981" stopOpacity={0} />

</linearGradient>

{*/\* Additional gradient definitions... \*/*}

</defs>

<CartesianGrid strokeDasharray="3 3" className="stroke-muted" />

<XAxis dataKey="name" className="text-xs" />

<YAxis className="text-xs" />

<Tooltip

content={({ active, payload, label }) => {

if (active && payload && payload.length) {

return (

<div className="rounded-lg border bg-background p-2 shadow-sm">

<div className="font-bold">{label}</div>

{payload.map((entry) => (

<div key={entry.name} className="flex items-center text-sm" style={{ color: entry.color }}>

<span className="mr-2 h-2 w-2 rounded-full" style={{ backgroundColor: entry.color }} />

<span>

{entry.name}: {entry.value}

</span>

</div>

))}

</div>

)

}

return null

}}

/>

{*/\* Chart areas... \*/*}

</AreaChart>

</ResponsiveContainer>

**Skeleton Loaders for Progressive Loading**

export function DashboardSkeleton() {

return (

<div className="space-y-6">

{*/\* Stats skeleton \*/*}

<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">

{Array(4)

.fill(0)

.map((\_, i) => (

<Card key={i}>

<CardHeader className="flex flex-row items-center justify-between pb-2">

<Skeleton className="h-5 w-24" />

<Skeleton className="h-8 w-8 rounded-full" />

</CardHeader>

<CardContent>

<Skeleton className="h-8 w-16 mb-2" />

<Skeleton className="h-4 w-32" />

</CardContent>

</Card>

))}

</div>

{*/\* Additional skeleton elements... \*/*}

</div>

)

}

**Timeline with Visual Indicators**

<div className="relative space-y-8 before:absolute before:inset-0 before:left-3.5 before:w-0.5 before:bg-muted before:ml-0.5">

{timeline.map((event, index) => (

<div key={index} className="relative pl-8">

<div

className={`absolute left-0 top-1 w-7 h-7 rounded-full flex items-center justify-center ${

event.status === "success"

? "bg-green-100 text-green-600 dark:bg-green-900 dark:text-green-400"

: event.status === "warning"

? "bg-amber-100 text-amber-600 dark:bg-amber-900 dark:text-amber-400"

: event.status === "error"

? "bg-red-100 text-red-600 dark:bg-red-900 dark:text-red-400"

: "bg-blue-100 text-blue-600 dark:bg-blue-900 dark:text-blue-400"

}`}

>

<div

className={`w-3 h-3 rounded-full ${

event.status === "success"

? "bg-green-500"

: event.status === "warning"

? "bg-amber-500"

: event.status === "error"

? "bg-red-500"

: "bg-blue-500"

}`}

></div>

</div>

{*/\* Event details... \*/*}

</div>

))}

</div>

**8. Known Issues and Future Development**

**Known Issues**

1. **Authentication System**:
   * Not fully implemented
   * Lacks sign-in/sign-up flows
   * Missing role-based access control
2. **Map Integration**:
   * Currently uses placeholder map UI
   * Needs integration with a real mapping service (Google Maps, Mapbox)
3. **Data Management**:
   * Create/Update/Delete operations not fully implemented
   * Form validation missing for data entry
4. **Neutrosophic AHP-TOPSIS Model**:
   * Core decision-making algorithm not fully implemented
   * Risk assessment calculations are placeholders

**Future Development**

**Short-term Priorities**

1. **Complete Authentication System**:
   * Implement sign-in/sign-up flows
   * Add role-based access control
   * Integrate with Supabase Auth
2. **Real Map Integration**:
   * Integrate with Mapbox or Google Maps
   * Implement real-time location tracking
   * Add route visualization
3. **CRUD Operations**:
   * Complete forms for creating and editing shipments
   * Implement validation and error handling
   * Add confirmation dialogs for destructive actions

**Medium-term Goals**

1. **Neutrosophic AHP-TOPSIS Implementation**:
   * Develop the core decision-making algorithm
   * Integrate with the frontend for real-time recommendations
   * Add visualization of decision factors
2. **Advanced Analytics**:
   * Predictive analytics for risk assessment
   * Machine learning integration for route optimization
   * More comprehensive reporting tools
3. **Notification System**:
   * Real-time alerts for critical events
   * Email and push notification integration
   * Customizable alert thresholds

**Long-term Vision**

1. **AI Integration**:
   * Predictive risk modeling
   * Automated decision support
   * Natural language processing for commands
2. **Mobile Application**:
   * Native mobile apps for iOS and Android
   * Offline-first architecture
   * Location services integration
3. **IoT Integration**:
   * Connect with tracking devices
   * Sensor data integration
   * Automated status updates

**9. Testing and Deployment**

**Testing Strategy**

While not currently implemented, the following testing strategy is recommended:

**Unit Testing**

* **Framework**: Jest with React Testing Library
* **Coverage**: Components, utilities, and hooks
* **Focus**: Component rendering, state management, and user interactions

*// Example unit test for DashboardStats component*

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

import { DashboardStats } from './dashboard-stats'

describe('DashboardStats', () => {

const mockStats = {

total\_shipments: 100,

active\_shipments: 50,

high\_risk\_shipments: 10,

delayed\_shipments: 5,

}

it('renders all stat cards correctly', () => {

render(<DashboardStats stats={mockStats} />)

expect(screen.getByText('Total Shipments')).toBeInTheDocument()

expect(screen.getByText('100')).toBeInTheDocument()

expect(screen.getByText('Active Shipments')).toBeInTheDocument()

expect(screen.getByText('50')).toBeInTheDocument()

expect(screen.getByText('High Risk')).toBeInTheDocument()

expect(screen.getByText('10')).toBeInTheDocument()

expect(screen.getByText('Delayed')).toBeInTheDocument()

expect(screen.getByText('5')).toBeInTheDocument()

})

})

**Integration Testing**

* **Framework**: Cypress
* **Coverage**: Page flows, data fetching, and user journeys
* **Focus**: Component interactions, API integration, and state management

*// Example Cypress integration test*

describe('Dashboard Page', () => {

beforeEach(() => {

cy.intercept('GET', '/api/dashboard/stats', { fixture: 'dashboard-stats.json' }).as('getStats')

cy.intercept('GET', '/api/shipments/recent', { fixture: 'recent-shipments.json' }).as('getShipments')

cy.intercept('GET', '/api/forwarders/performance', { fixture: 'forwarder-performance.json' }).as('getForwarders')

cy.visit('/')

cy.wait(['@getStats', '@getShipments', '@getForwarders'])

})

it('displays dashboard components correctly', () => {

cy.get('[data-testid="dashboard-header"]').should('be.visible')

cy.get('[data-testid="dashboard-stats"]').should('be.visible')

cy.get('[data-testid="shipment-risk-chart"]').should('be.visible')

cy.get('[data-testid="forwarder-performance"]').should('be.visible')

cy.get('[data-testid="recent-shipments"]').should('be.visible')

})

it('allows filtering by time period', () => {

cy.get('[data-testid="time-period-select"]').click()

cy.get('[data-value="week"]').click()

cy.wait(['@getStats', '@getShipments', '@getForwarders'])

*// Verify filtered data is displayed*

})

})

**End-to-End Testing**

* **Framework**: Playwright
* **Coverage**: Complete user flows across multiple pages
* **Focus**: Real-world scenarios, edge cases, and performance

*// Example Playwright E2E test*

import { test, expect } from '@playwright/test'

test('complete shipment tracking flow', async ({ page }) => {

*// Log in*

await page.goto('/login')

await page.fill('[data-testid="email-input"]', 'test@example.com')

await page.fill('[data-testid="password-input"]', 'password')

await page.click('[data-testid="login-button"]')

*// Navigate to shipments*

await page.click('[data-testid="shipments-nav"]')

await expect(page).toHaveURL('/shipments')

*// Search for a shipment*

await page.fill('[data-testid="search-input"]', 'REQ-12345')

await page.press('[data-testid="search-input"]', 'Enter')

*// Click on a shipment*

await page.click('text=REQ-12345')

await expect(page).toHaveURL(/\/shipments\/.\*/)

*// Verify shipment details are displayed*

await expect(page.locator('[data-testid="shipment-details"]')).toBeVisible()

await expect(page.locator('[data-testid="shipment-timeline"]')).toBeVisible()

await expect(page.locator('[data-testid="shipment-map"]')).toBeVisible()

})

**Deployment Process**

**Development Environment**

* **Local Development**: Next.js development server
* **Database**: Local Supabase instance or development project

**Staging Environment**

* **Hosting**: Vercel Preview Deployments
* **Database**: Staging Supabase project
* **CI/CD**: Automated deployments from feature branches

**Production Environment**

* **Hosting**: Vercel Production Deployment
* **Database**: Production Supabase project
* **CI/CD**: Automated deployments from main branch with approval

**Deployment Configuration**

*// vercel.json*

{

"version": 2,

"builds": [

{

"src": "package.json",

"use": "@vercel/next"

}

],

"routes": [

{

"src": "/(.\*)",

"dest": "/$1"

}

],

"env": {

"NEXT\_PUBLIC\_SUPABASE\_URL": "@supabase\_url",

"NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY": "@supabase\_anon\_key"

}

}

**10. Conclusion**

DeepCAL represents a sophisticated emergency logistics decision-support system with a modern, responsive interface and real-time capabilities. The application successfully implements:

* A comprehensive dashboard for monitoring key metrics
* Real-time shipment tracking with geospatial visualization
* Detailed shipment management capabilities
* Performance evaluation of logistics partners
* Interactive analytics for data-driven insights

The application's architecture follows best practices with:

* Clear separation of concerns
* Type-safe data handling
* Responsive and accessible UI
* Real-time data updates
* Progressive loading for better user experience

While there are areas that require further development, particularly the authentication system, map integration, and the core Neutrosophic AHP-TOPSIS decision model, the current implementation provides a solid foundation for an intelligent emergency logistics platform.

The SuperFuturistic UI/UX design approach, with animated components, micro-interactions, and a clean, information-dense interface, prioritizes usability while maintaining visual appeal, making DeepCAL an effective tool for emergency response coordinators and logistics managers who need to make rapid, data-driven decisions in high-pressure situations.

I hope the outline is based on these

DeepCAL: Intelligent Logistics and Analytics Dashboard Table of Contents Introduction

System Architecture

UI Pages & Component Breakdown

3.1 Mapview (Main Page)

3.2 Analytics (Shipment, Destination, Freight, Deep Insights)

3.3 DeepCAL Calculator

3.4 Settings

Data Integration & APIs

Database Schema

Security & Performance Considerations

Conclusion

1. Introduction DeepCAL is an AI-powered logistics and analytics platform optimized for emergency response and supply chain efficiency. The system integrates real-time shipment tracking, freight forwarder evaluation, and deep analytical insights, leveraging the Neutrosophic AHP-TOPSIS model for intelligent decision-making.

This documentation outlines the interface structure, components, and data flow of DeepCAL, ensuring a well-defined, scalable, and intuitive user experience.

1. System Architecture DeepCAL follows a modular design, consisting of:

Frontend (Next.js, Tailwind CSS, shadcn/ui) for an interactive user interface.

Backend (FastAPI, Python-based decision engine) for complex analytics and AI computations.

Database (Supabase - PostgreSQL) for data storage, real-time updates, and API services.

Edge Functions (Supabase serverless logic) for background processing and automation.

Each module interacts seamlessly through well-defined REST APIs.

1. UI Pages & Component Breakdown 3.1 Mapview (Main Page) Description A full-screen interactive map for shipment tracking, geospatial risk assessment, and logistics optimization.

Key Features Live Shipment Tracking: Displays real-time shipment locations using Mapbox GL.

Geospatial Risk Indicators: Color-coded overlays for high-risk areas based on real-time data.

Interactive Filtering: Users can toggle shipment categories (e.g., high-priority, delayed, at-risk).

Route Optimization: AI-powered recommendations for alternate routing based on risk analytics.

Component Breakdown Component Functionality Map Widget Displays real-time locations, routes, and heatmaps. Risk Heatmap Highlights high-risk zones for shipments. Filter Controls Allow users to refine displayed shipments by priority, status, etc. Detailed Shipment View Clickable markers provide shipment details, including risk scores. 3.2 Analytics Page Description A dashboard containing multiple analytic insights into shipment performance, risk factors, and freight efficiency.

Tabs & Insights Tab Features Shipment Analytics Graphs and tables on delivery times, cost efficiency, and real-time shipment data. Destination Analytics Evaluates risk levels and delivery success rates across destinations. Freight Analytics Performance and reliability scores of freight forwarders. Deep Insights AI-driven predictions on shipment risks, cost variations, and supply chain bottlenecks. Component Breakdown Component Functionality Bar & Line Charts Displays historical and predictive shipment trends. Performance Scorecards Summarizes key logistics metrics (OTIF %, delay rates, cost deviation). Interactive Data Tables Allows filtering and deep dives into logistics performance. Heatmap View Geospatial visualization of high-risk destinations. 3.3 DeepCAL Calculator Description A real-time logistics decision engine that computes the best freight options, shipment routes, and risk adjustments.

Key Features Neutrosophic AHP-TOPSIS Computation: Uses AI-based ranking models to evaluate shipment alternatives.

Automated Risk Assessment: Dynamically adjusts weightings based on historical and real-time factors.

Cost-Performance Optimization: Recommends the most efficient routes and freight forwarders.

Component Breakdown Component Functionality Decision Engine UI Allows users to input shipment criteria and run calculations. Dynamic Weight Sliders Adjusts factor importance (cost, reliability, responsiveness). Ranking Table Displays optimal options ranked based on AHP-TOPSIS. AI Insights Provides reasoning and recommendations for selected shipment plans. 3.4 Settings Description A configuration panel for managing DeepCAL system preferences, user roles, API integrations, and visual themes.

Key Features General System Settings: Manage analytics update frequency, alert thresholds, and API integrations.

User Role Management: Define access control levels for different stakeholders.

Customization & Themes: Adapt UI themes (light/dark mode), adjust dashboard layout.

API & Integration Management: Configure connections with Supabase, Mapbox, OpenWeather API, etc.

1. Data Integration & APIs DeepCAL integrates with multiple external services and APIs for real-time insights.

API Function Supabase Database API Handles storage of shipment, freight, and risk data. Mapbox API Provides real-time mapping, geospatial risk visualization. OpenWeather API Supplies weather conditions for risk assessment. DeepCAL Decision Engine (FastAPI) Runs AHP-TOPSIS ranking and risk analysis. 5. Database Schema DeepCAL utilizes a normalized PostgreSQL schema in Supabase.

Shipment Table sql Copy Edit CREATE TABLE shipments ( id SERIAL PRIMARY KEY, origin VARCHAR(255), destination VARCHAR(255), weight FLOAT, volume FLOAT, cost FLOAT, status VARCHAR(50), estimated\_arrival TIMESTAMP ); Freight Forwarders Table sql Copy Edit CREATE TABLE freight\_forwarders ( id SERIAL PRIMARY KEY, name VARCHAR(255), service\_type VARCHAR(255), cost\_metric FLOAT, reliability\_score FLOAT ); Risk Assessment Table sql Copy Edit CREATE TABLE risk\_assessment ( id SERIAL PRIMARY KEY, shipment\_id INT REFERENCES shipments(id), risk\_score FLOAT, delay\_probability FLOAT, reason TEXT ); API Request Logging sql Copy Edit CREATE TABLE api\_logs ( id SERIAL PRIMARY KEY, endpoint VARCHAR(255), request\_time TIMESTAMP, response\_time FLOAT, status\_code INT ); 6. Security & Performance Considerations Authentication: Uses Supabase Auth (JWT-based).

Rate Limiting: Prevents excessive API calls.

Caching Strategies: Edge functions optimize data retrieval to improve responsiveness.

Monitoring & Logging: Logs API calls and performance metrics for system health tracking.

1. Conclusion DeepCAL is an advanced AI-powered logistics analytics and decision-making tool, integrating real-time tracking, predictive risk assessment, and automated freight optimization. Its architecture supports scalability, security, and seamless integration with modern APIs and decision-making models.

This document serves as the foundation for accurate implementation and development, ensuring that DeepCAL provides an intuitive and data-driven interface for efficient emergency logistics and supply chain management.

Thought for 3 seconds

**DeepCAL: Technical Documentation Report**

**Table of Contents**

1. [Overview](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#1-overview)
2. [Architecture](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#2-architecture)
3. [File Structure](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#3-file-structure)
4. [Components and Functionalities](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#4-components-and-functionalities)
   * 4.1 [Dashboard](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#41-dashboard)
   * 4.2 [Live Tracking](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#42-live-tracking)
   * 4.3 [Shipments Management](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#43-shipments-management)
   * 4.4 [Forwarders Management](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#44-forwarders-management)
   * 4.5 [Analytics](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#45-analytics)
5. [Supabase Integration](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#5-supabase-integration)
6. [Data Flow](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#6-data-flow)
7. [Styling and UI/UX](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#7-styling-and-uiux)
8. [Known Issues and Future Development](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#8-known-issues-and-future-development)
9. [Testing and Deployment](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#9-testing-and-deployment)
10. [API Documentation](https://v0.dev/chat/deepcal-frontend-implementation-BMnsnq3tk8P#10-api-documentation)

**1. Overview**

DeepCAL (Deep Crisis Analytics and Logistics) is an intelligent emergency logistics platform designed to optimize the allocation and delivery of aid and resources during crises. The application serves as a comprehensive decision-support system for emergency response teams, logistics coordinators, and aid organizations.

**Purpose**

DeepCAL's primary purpose is to:

* Provide real-time tracking and monitoring of emergency shipments
* Assess and visualize risk levels for shipments and routes
* Evaluate and compare logistics partners (forwarders)
* Offer data-driven insights through interactive analytics
* Support decision-making through the Neutrosophic AHP-TOPSIS model

**Target Users**

* Emergency response coordinators
* Logistics managers in humanitarian organizations
* Supply chain professionals in crisis management
* Aid distribution teams
* Government emergency management agencies

**Key Features**

* Real-time shipment tracking with geospatial visualization
* Risk assessment and prioritization of shipments
* Performance evaluation of logistics partners
* Comprehensive analytics dashboard
* Interactive data tables and visualizations
* Dark/light mode support for different operational environments

**2. Architecture**

DeepCAL follows a modern, modular architecture leveraging cutting-edge web technologies to deliver a responsive, real-time application.

**Technology Stack**

**Frontend**

* **Next.js**: React framework with App Router for server components and routing
* **TypeScript**: For type-safe code and improved developer experience
* **Tailwind CSS**: For utility-first styling and responsive design
* **shadcn/ui**: Component library built on Radix UI primitives
* **Recharts**: For interactive data visualization
* **TanStack Table**: For advanced table functionality

**Backend**

* **Supabase**: Backend-as-a-Service platform providing:
  + PostgreSQL database
  + Authentication services
  + Real-time subscriptions
  + Edge Functions (planned)
* **Server Components**: Next.js server components for server-side rendering and data fetching
* **Server Actions**: For handling form submissions and data mutations

**Decision Engine (Planned)**

* **Neutrosophic AHP-TOPSIS Model**: For multi-criteria decision-making
* **FastAPI**: For Python-based microservices (planned)

**Architecture Diagram**

DeepCAL Architecture

Client BrowserNext.js FrontendNext.js Server ComponentsClient ComponentsSupabase Client (Server)Supabase Client (Browser)Supabase BackendPostgreSQL DatabaseReal-time SubscriptionsAuthenticationDecision Engine (Planned)Neutrosophic AHP-TOPSIS Model

**Component Interaction**

* **Server Components** fetch data directly from Supabase during server-side rendering
* **Client Components** use Supabase client for real-time updates and interactive features
* **Real-time Subscriptions** enable live updates without manual refreshing
* **Server Actions** handle form submissions and data mutations
* **Decision Engine** (planned) will process complex logistics calculations and risk assessments

**3. File Structure**

DeepCAL follows a well-organized file structure that aligns with Next.js App Router conventions and promotes separation of concerns.

**Root Directory Structure**

deepcal/

├── app/ # Next.js App Router pages and layouts

├── components/ # Reusable UI components

├── lib/ # Utility functions and type definitions

├── public/ # Static assets

├── styles/ # Global styles (if any)

├── next.config.js # Next.js configuration

├── tailwind.config.js # Tailwind CSS configuration

├── tsconfig.json # TypeScript configuration

└── package.json # Project dependencies

**App Directory**

app/

├── layout.tsx # Root layout with providers

├── page.tsx # Dashboard page (home)

├── globals.css # Global styles and Tailwind directives

├── live-tracking/ # Live tracking route

│ └── page.tsx

├── shipments/ # Shipments management routes

│ ├── page.tsx

│ └── [id]/ # Dynamic route for shipment details

│ └── page.tsx

├── forwarders/ # Forwarders management route

│ └── page.tsx

└── analytics/ # Analytics dashboard route

└── page.tsx

**Components Directory**

components/

├── ui/ # shadcn/ui components

│ ├── sidebar.tsx

│ ├── button.tsx

│ ├── card.tsx

│ ├── chart.tsx

│ └── ...

├── app-sidebar.tsx # Application sidebar

├── mode-toggle.tsx # Dark/light mode toggle

├── theme-provider.tsx # Theme provider component

├── dashboard/ # Dashboard components

│ ├── dashboard-header.tsx

│ ├── dashboard-stats.tsx

│ ├── shipment-risk-chart.tsx

│ ├── forwarder-performance.tsx

│ ├── recent-shipments.tsx

│ └── dashboard-skeleton.tsx

├── live-tracking/ # Live tracking components

│ ├── live-tracking-header.tsx

│ ├── live-tracking-map.tsx

│ ├── live-shipment-list.tsx

│ └── shipment-details-sidebar.tsx

├── shipments/ # Shipments components

│ ├── shipments-header.tsx

│ ├── shipments-table.tsx

│ ├── shipments-data-table.tsx

│ ├── shipment-details.tsx

│ └── ...

├── forwarders/ # Forwarders components

│ ├── forwarders-header.tsx

│ ├── forwarders-grid.tsx

│ └── forwarder-card.tsx

└── analytics/ # Analytics components

├── analytics-header.tsx

├── analytics-overview.tsx

├── shipment-trends-chart.tsx

└── ...

**Library Directory**

lib/

├── supabase.ts # Supabase client initialization

├── database.types.ts # TypeScript types for database schema

└── utils.ts # Utility functions

**Key Files**

**Root Layout (**app/layout.tsx**)**

export default function RootLayout({

children,

}: {

children: React.ReactNode

}) {

return (

<html lang="en" suppressHydrationWarning>

<body className={inter.className}>

<ThemeProvider attribute="class" defaultTheme="system" enableSystem disableTransitionOnChange>

<SidebarProvider>

<div className="flex h-screen">

<AppSidebar />

<div className="flex-1 overflow-auto">{children}</div>

</div>

</SidebarProvider>

</ThemeProvider>

</body>

</html>

)

}

**Supabase Client (**lib/supabase.ts**)**

import { createClient } from "@supabase/supabase-js"

import type { Database } from "@/lib/database.types"

const supabaseUrl = process.env.NEXT\_PUBLIC\_SUPABASE\_URL!

const supabaseAnonKey = process.env.NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY!

export const supabase = createClient<Database>(supabaseUrl, supabaseAnonKey)

*// Server-side client for use in Server Components and API routes*

export const createServerSupabaseClient = async () => {

const { cookies } = await import("next/headers")

const cookieStore = cookies()

return createClient<Database>(supabaseUrl, supabaseAnonKey, {

cookies: {

get(name: string) {

return cookieStore.get(name)?.value

},

},

})

}

**Database Types (**lib/database.types.ts**)**

export type Json = string | number | boolean | null | { [key: string]: Json | undefined } | Json[]

export interface Database {

public: {

Tables: {

shipments: {

Row: {

id: string

created\_at: string

updated\_at: string

request\_reference: string

origin: string

destination: string

status: string

risk\_level: number

estimated\_arrival: string

actual\_arrival: string | null

forwarder\_id: string

customer\_id: string

priority: "low" | "medium" | "high" | "critical"

location: { lat: number; lng: number } | null

}

*// Insert and Update types omitted for brevity*

}

forwarders: {

Row: {

id: string

created\_at: string

name: string

performance\_score: number

active\_shipments: number

total\_shipments: number

on\_time\_rate: number

}

*// Insert and Update types omitted for brevity*

}

users: {

Row: {

id: string

email: string

role: "admin" | "operator" | "viewer"

name: string

}

*// Insert and Update types omitted for brevity*

}

}

}

}

**4. Components and Functionalities**

**4.1 Dashboard**

The Dashboard serves as the application's home page, providing a comprehensive overview of emergency logistics operations.

**Dashboard Page (**app/page.tsx**)**

export default async function DashboardPage() {

return (

<div className="flex flex-col gap-6 p-6 md:p-8">

<DashboardHeader />

<Suspense fallback={<DashboardSkeleton />}>

<DashboardContent />

</Suspense>

</div>

)

}

async function DashboardContent() {

const supabase = await createServerSupabaseClient()

*// Fetch summary statistics*

const { data: stats } = await supabase.rpc("get\_dashboard\_stats")

*// Fetch recent shipments*

const { data: recentShipments } = await supabase

.from("shipments")

.select(`

id,

request\_reference,

origin,

destination,

status,

risk\_level,

estimated\_arrival,

priority,

forwarders(name)

`)

.order("created\_at", { ascending: false })

.limit(5)

*// Fetch forwarder performance*

const { data: forwarders } = await supabase

.from("forwarders")

.select("\*")

.order("performance\_score", { ascending: false })

.limit(5)

return (

<>

<DashboardStats stats={stats || {...}} />

<div className="grid grid-cols-1 lg:grid-cols-2 gap-6">

<ShipmentRiskChart />

<ForwarderPerformance forwarders={forwarders || []} />

</div>

<RecentShipments shipments={recentShipments || []} />

</>

)

}

**Key Components**

1. **Dashboard Header** (components/dashboard/dashboard-header.tsx)
   * Title and description
   * Time period selector
   * Refresh and export buttons
2. **Dashboard Stats** (components/dashboard/dashboard-stats.tsx)
   * Key performance indicators:
     + Total Shipments
     + Active Shipments
     + High Risk Shipments
     + Delayed Shipments
   * Each stat includes trend indicators
3. **Shipment Risk Chart** (components/dashboard/shipment-risk-chart.tsx)
   * Area chart showing distribution of shipments by risk level
   * Interactive tooltips with detailed information
   * Color-coded risk levels (low, medium, high, critical)
4. **Forwarder Performance** (components/dashboard/forwarder-performance.tsx)
   * Bar chart comparing logistics partners
   * Performance scores and on-time delivery rates
   * Interactive tooltips with detailed metrics
5. **Recent Shipments** (components/dashboard/recent-shipments.tsx)
   * Table of latest shipment activity
   * Status badges and risk indicators
   * Quick links to shipment details

**4.2 Live Tracking**

The Live Tracking page provides real-time monitoring of active shipments with geospatial visualization.

**Live Tracking Page (**app/live-tracking/page.tsx**)**

export default function LiveTrackingPage() {

return (

<div className="flex flex-col h-screen">

<div className="p-6 md:p-8">

<LiveTrackingHeader />

</div>

<div className="flex-1 flex flex-col lg:flex-row gap-6 p-6 md:px-8 md:pb-8 pt-0">

<Suspense fallback={<Skeleton className="h-[calc(100vh-200px)] w-full rounded-lg" />}>

<LiveTrackingMap />

</Suspense>

<div className="w-full lg:w-96 flex flex-col gap-6">

<Suspense fallback={<Skeleton className="h-[calc(100vh-200px)] w-full rounded-lg" />}>

<LiveShipmentList />

</Suspense>

</div>

</div>

<ShipmentDetailsSidebar />

</div>

)

}

**Key Components**

1. **Live Tracking Map** (components/live-tracking/live-tracking-map.tsx)
   * Interactive map showing shipment locations
   * Real-time updates via Supabase subscriptions
   * Color-coded markers based on risk level
   * Map controls and legend
2. **Live Shipment List** (components/live-tracking/live-shipment-list.tsx)
   * List of active shipments with real-time updates
   * Risk indicators and priority badges
   * Origin-destination information
   * Quick links to shipment details
3. **Shipment Details Sidebar** (components/live-tracking/shipment-details-sidebar.tsx)
   * Slide-in panel with comprehensive shipment information
   * Tabbed interface (Details, Timeline, Contacts)
   * Status indicators and risk assessment
   * Action buttons for shipment management

**4.3 Shipments Management**

The Shipments Management section provides comprehensive tools for managing and monitoring all shipments in the system.

**Shipments Page (**app/shipments/page.tsx**)**

export default function ShipmentsPage() {

return (

<div className="flex flex-col gap-6 p-6 md:p-8">

<ShipmentsHeader />

<Suspense fallback={<ShipmentsTableSkeleton />}>

<ShipmentsTable />

</Suspense>

</div>

)

}

**Shipment Details Page (**app/shipments/[id]/page.tsx**)**

export default async function ShipmentPage({ params }: ShipmentPageProps) {

return (

<div className="flex flex-col gap-6 p-6 md:p-8">

<Suspense fallback={<Skeleton className="h-12 w-64" />}>

<ShipmentHeader id={params.id} />

</Suspense>

<div className="grid grid-cols-1 lg:grid-cols-3 gap-6">

<div className="lg:col-span-2 space-y-6">

<Suspense fallback={<Skeleton className="h-[300px] w-full rounded-lg" />}>

<ShipmentMap id={params.id} />

</Suspense>

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<ShipmentTimeline id={params.id} />

</Suspense>

</div>

<div className="space-y-6">

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<ShipmentDetails id={params.id} />

</Suspense>

<Suspense fallback={<Skeleton className="h-[100px] w-full rounded-lg" />}>

<ShipmentActions id={params.id} />

</Suspense>

</div>

</div>

</div>

)

}

**Key Components**

1. **Shipments Table** (components/shipments/shipments-table.tsx)
   * Server component that fetches shipment data
   * Passes data to client-side data table component
2. **Shipments Data Table** (components/shipments/shipments-data-table.tsx)
   * Interactive table with sorting, filtering, and pagination
   * Status badges and risk indicators
   * Action menu for each shipment
   * Column visibility toggle
3. **Shipment Details** (components/shipments/shipment-details.tsx)
   * Comprehensive information about a specific shipment
   * Basic information, route details, time information
   * Risk assessment visualization
   * Forwarder and customer information
4. **Shipment Timeline** (components/shipments/shipment-timeline.tsx)
   * Chronological view of shipment events
   * Status indicators for each event
   * Time and location information
   * Visual timeline with connecting lines
5. **Shipment Map** (components/shipments/shipment-map.tsx)
   * Map showing the current location of a specific shipment
   * Real-time updates via Supabase subscriptions
   * Map controls and location information
6. **Shipment Actions** (components/shipments/shipment-actions.tsx)
   * Action buttons for shipment management
   * Contact forwarder, send message, refresh status
   * Flag issue, mark delivered, cancel shipment

**4.4 Forwarders Management**

The Forwarders Management section provides tools for evaluating and managing logistics partners.

**Forwarders Page (**app/forwarders/page.tsx**)**

export default function ForwardersPage() {

return (

<div className="flex flex-col gap-6 p-6 md:p-8">

<ForwardersHeader />

<Suspense fallback={<ForwardersGridSkeleton />}>

<ForwardersGrid />

</Suspense>

</div>

)

}

**Key Components**

1. **Forwarders Grid** (components/forwarders/forwarders-grid.tsx)
   * Grid of forwarder cards
   * Fetches forwarder data from Supabase
2. **Forwarder Card** (components/forwarders/forwarder-card.tsx)
   * Card displaying forwarder information
   * Performance score and on-time rate
   * Active and total shipments
   * Action menu and quick links

**4.5 Analytics**

The Analytics section provides comprehensive insights into logistics operations with interactive visualizations.

**Analytics Page (**app/analytics/page.tsx**)**

export default function AnalyticsPage() {

return (

<div className="flex flex-col gap-6 p-6 md:p-8">

<AnalyticsHeader />

<Suspense fallback={<Skeleton className="h-[120px] w-full rounded-lg" />}>

<AnalyticsOverview />

</Suspense>

<div className="grid grid-cols-1 lg:grid-cols-2 gap-6">

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<ShipmentTrendsChart />

</Suspense>

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<RiskDistributionChart />

</Suspense>

</div>

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<ForwarderComparisonChart />

</Suspense>

<Suspense fallback={<Skeleton className="h-[400px] w-full rounded-lg" />}>

<PerformanceMetricsTable />

</Suspense>

</div>

)

}

**Key Components**

1. **Analytics Overview** (components/analytics/analytics-overview.tsx)
   * Key performance indicators with trend indicators
   * Total shipments, on-time deliveries, average risk score, active forwarders
2. **Shipment Trends Chart** (components/analytics/shipment-trends-chart.tsx)
   * Line chart showing shipment volumes and delivery performance over time
   * Total shipments, on-time deliveries, delayed shipments
3. **Risk Distribution Chart** (components/analytics/risk-distribution-chart.tsx)
   * Pie chart showing distribution of shipments by risk level
   * Low, medium, high, and critical risk categories
4. **Forwarder Comparison Chart** (components/analytics/forwarder-comparison-chart.tsx)
   * Bar chart comparing performance metrics across logistics partners
   * Performance score, on-time rate, cost efficiency
5. **Performance Metrics Table** (components/analytics/performance-metrics-table.tsx)
   * Table showing key performance indicators for major shipping routes
   * Average delivery time, on-time rate, risk score, incidents, cost per km

**5. Supabase Integration**

DeepCAL leverages Supabase as its backend-as-a-service platform for data storage, real-time updates, and authentication.

**Client Initialization**

*// lib/supabase.ts*

import { createClient } from "@supabase/supabase-js"

import type { Database } from "@/lib/database.types"

const supabaseUrl = process.env.NEXT\_PUBLIC\_SUPABASE\_URL!

const supabaseAnonKey = process.env.NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY!

*// Browser client for use in Client Components*

export const supabase = createClient<Database>(supabaseUrl, supabaseAnonKey)

*// Server-side client for use in Server Components and API routes*

export const createServerSupabaseClient = async () => {

const { cookies } = await import("next/headers")

const cookieStore = cookies()

return createClient<Database>(supabaseUrl, supabaseAnonKey, {

cookies: {

get(name: string) {

return cookieStore.get(name)?.value

},

},

})

}

**Data Fetching**

**Server-Side Data Fetching**

*// Example from components/shipments/shipments-table.tsx*

export async function ShipmentsTable() {

const supabase = await createServerSupabaseClient()

*// Fetch shipments with forwarder information*

const { data: shipments, error } = await supabase

.from("shipments")

.select(`

id,

request\_reference,

origin,

destination,

status,

risk\_level,

estimated\_arrival,

actual\_arrival,

priority,

created\_at,

forwarder\_id,

forwarders(name)

`)

.order("created\_at", { ascending: false })

if (error) {

console.error("Error fetching shipments:", error)

throw new Error("Failed to fetch shipments")

}

return <ShipmentsDataTable data={shipments || []} />

}

**Client-Side Data Fetching**

*// Example from components/live-tracking/live-tracking-map.tsx*

useEffect(() => {

*// Initial fetch of shipments with location data*

const fetchShipments = async () => {

try {

const { data, error } = await supabase

.from("shipments")

.select("\*")

.not("location", "is", null)

.order("risk\_level", { ascending: false })

if (error) throw error

setShipments(data || [])

} catch (error) {

console.error("Error fetching shipments:", error)

toast({

title: "Error fetching shipments",

description: "Could not load shipment location data",

variant: "destructive",

})

} finally {

setIsLoading(false)

}

}

fetchShipments()

*// ...*

}, [toast])

**Real-Time Subscriptions**

*// Example from components/live-tracking/live-tracking-map.tsx*

useEffect(() => {

*// Initial fetch code omitted for brevity...*

*// Subscribe to real-time updates*

const subscription = supabase

.channel("shipment-location-changes")

.on(

"postgres\_changes",

{

event: "UPDATE",

schema: "public",

table: "shipments",

filter: "location=neq.null",

},

(payload) => {

const updatedShipment = payload.new as Shipment

setShipments((current) => {

const index = current.findIndex((s) => s.id === updatedShipment.id)

if (index >= 0) {

const updated = [...current]

updated[index] = updatedShipment

return updated

} else {

return [...current, updatedShipment]

}

})

},

)

.subscribe()

return () => {

subscription.unsubscribe()

}

}, [toast])

**Authentication (Planned)**

While not fully implemented in the current codebase, the foundation for authentication is in place:

*// Planned authentication flow*

async function signIn(email: string, password: string) {

const { data, error } = await supabase.auth.signInWithPassword({

email,

password,

})

if (error) {

throw error

}

return data

}

async function signOut() {

const { error } = await supabase.auth.signOut()

if (error) {

throw error

}

}

**6. Data Flow**

DeepCAL implements a sophisticated data flow architecture that combines server-side rendering, client-side interactivity, and real-time updates.

**Data Flow Diagram**

DeepCAL Data Flow

User InteractionClient ComponentSupabase Client (Browser)Supabase BackendPostgreSQL DatabaseReal-time SubscriptionPage RequestServer ComponentSupabase Client (Server)Rendered HTMLHydration

**Server-Side Data Flow**

1. **Initial Page Request**:
   * User navigates to a page
   * Next.js server receives the request
   * Server Component is executed
2. **Server-Side Data Fetching**:
   * Server Component creates a Supabase client
   * Client fetches data from Supabase
   * Data is processed and passed to child components
3. **HTML Generation**:
   * Server Component renders HTML with data
   * HTML is sent to the browser
   * Client-side JavaScript is loaded
4. **Hydration**:
   * Client-side JavaScript takes over
   * Interactive elements become functional
   * Client Components are hydrated with server data

**Client-Side Data Flow**

1. **User Interaction**:
   * User interacts with the interface (click, input, etc.)
   * Client Component handles the interaction
2. **Client-Side Data Fetching**:
   * Client Component creates a Supabase client
   * Client fetches data from Supabase
   * Data is processed and used to update the UI
3. **Real-Time Updates**:
   * Client Component subscribes to Supabase channel
   * Supabase sends real-time updates when data changes
   * Client Component updates the UI with new data

**Example: Shipment Details Page**

1. User navigates to /shipments/123
2. Server fetches shipment data for ID "123"
3. Server renders the page with shipment details
4. Client hydrates the page, making it interactive
5. Client subscribes to real-time updates for shipment "123"
6. When shipment data changes, UI updates automatically

**7. Styling and UI/UX**

DeepCAL features a modern, responsive UI with a focus on clarity, accessibility, and futuristic design elements.

**Styling Approach**

The application uses Tailwind CSS for styling, with a custom color scheme defined using CSS variables:

*/\* From globals.css \*/*

@layer base {

:root {

--background: 0 0% 100%;

--foreground: 240 10% 3.9%;

--card: 0 0% 100%;

--card-foreground: 240 10% 3.9%;

--popover: 0 0% 100%;

--popover-foreground: 240 10% 3.9%;

--primary: 240 5.9% 10%;

--primary-foreground: 0 0% 98%;

--secondary: 240 4.8% 95.9%;

--secondary-foreground: 240 5.9% 10%;

--muted: 240 4.8% 95.9%;

--muted-foreground: 240 3.8% 46.1%;

--accent: 240 4.8% 95.9%;

--accent-foreground: 240 5.9% 10%;

--destructive: 0 84.2% 60.2%;

--destructive-foreground: 0 0% 98%;

--border: 240 5.9% 90%;

--input: 240 5.9% 90%;

--ring: 240 5.9% 10%;

--radius: 0.5rem;

*/\* Sidebar variables \*/*

--sidebar-background: 0 0% 98%;

--sidebar-foreground: 240 5.3% 26.1%;

*/\* Additional sidebar variables omitted for brevity \*/*

}

.dark {

--background: 240 10% 3.9%;

--foreground: 0 0% 98%;

*/\* Dark mode variables omitted for brevity \*/*

}

}

**Component Library**

The application uses shadcn/ui components, which are built on Radix UI primitives and styled with Tailwind CSS. These components provide:

* Accessibility out of the box
* Consistent styling and behavior
* Customizability through Tailwind classes
* Dark/light mode support

**SuperFuturistic UI/UX Elements**

**Animated Logo**

<div className="relative w-8 h-8">

<div className="absolute inset-0 bg-gradient-to-br from-primary to-indigo-600 rounded-lg animate-pulse"></div>

<div className="absolute inset-0.5 bg-background rounded-lg flex items-center justify-center">

<span className="font-bold text-primary">DC</span>

</div>

</div>

**Interactive Charts**

The application uses Recharts for data visualization, with custom styling and interactive features:

<ResponsiveContainer width="100%" height="100%">

<AreaChart

data={riskData}

margin={{ top: 10, right: 10, left: 0, bottom: 0 }}

onMouseMove={(e) => {

if (e.activeTooltipIndex !== undefined) {

setActiveIndex(e.activeTooltipIndex)

}

}}

onMouseLeave={() => setActiveIndex(null)}

>

<defs>

<linearGradient id="colorLow" x1="0" y1="0" x2="0" y2="1">

<stop offset="5%" stopColor="#10b981" stopOpacity={0.8} />

<stop offset="95%" stopColor="#10b981" stopOpacity={0} />

</linearGradient>

{*/\* Additional gradients omitted for brevity \*/*}

</defs>

<CartesianGrid strokeDasharray="3 3" className="stroke-muted" />

<XAxis dataKey="name" className="text-xs" />

<YAxis className="text-xs" />

<Tooltip content={CustomTooltip} />

<Area

type="monotone"

dataKey="low"

stackId="1"

stroke="#10b981"

fillOpacity={1}

fill="url(#colorLow)"

name="Low Risk"

/>

{*/\* Additional areas omitted for brevity \*/*}

</AreaChart>

</ResponsiveContainer>

**Skeleton Loaders**

The application uses skeleton loaders for a smoother perceived performance:

export function DashboardSkeleton() {

return (

<div className="space-y-6">

{*/\* Stats skeleton \*/*}

<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">

{Array(4)

.fill(0)

.map((\_, i) => (

<Card key={i}>

<CardHeader className="flex flex-row items-center justify-between pb-2">

<Skeleton className="h-5 w-24" />

<Skeleton className="h-8 w-8 rounded-full" />

</CardHeader>

<CardContent>

<Skeleton className="h-8 w-16 mb-2" />

<Skeleton className="h-4 w-32" />

</CardContent>

</Card>

))}

</div>

{*/\* Additional skeleton elements omitted for brevity \*/*}

</div>

)

}

**Status Indicators**

The application uses color-coded badges and indicators to convey status information:

const getRiskBadge = (risk: number) => {

if (risk < 25) return <Badge className="bg-green-500">Low</Badge>

if (risk < 50) return <Badge className="bg-amber-500">Medium</Badge>

if (risk < 75) return <Badge className="bg-red-500">High</Badge>

return <Badge className="bg-red-900">Critical</Badge>

}

const getStatusBadge = (status: string) => {

switch (status.toLowerCase()) {

case "in transit":

return (

<Badge variant="outline" className="border-blue-500 text-blue-500">

In Transit

</Badge>

)

case "delivered":

return (

<Badge variant="outline" className="border-green-500 text-green-500">

Delivered

</Badge>

)

*// Additional cases omitted for brevity*

}

}

**Responsive Design**

The application is fully responsive, with adaptive layouts for different screen sizes:

*// Example of responsive grid*

<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-6">

{*/\* Content \*/*}

</div>

*// Example of responsive flex layout*

<div className="flex flex-col lg:flex-row gap-6">

{*/\* Content \*/*}

</div>

**8. Known Issues and Future Development**

**Known Issues**

1. **Map Integration**:
   * Currently uses placeholder map UI
   * Needs integration with a real mapping service (Google Maps, Mapbox)
2. **Authentication System**:
   * Not fully implemented
   * Lacks sign-in/sign-up flows
   * Missing role-based access control
3. **Form Validation**:
   * Limited validation for user inputs
   * No comprehensive error handling for form submissions
4. **Data Management**:
   * Create/Update/Delete operations not fully implemented
   * Limited data validation on the client side

**Future Development**

**Short-term Priorities**

1. **Complete Authentication System**:
   * Implement sign-in/sign-up flows
   * Add role-based access control
   * Integrate with Supabase Auth
2. **Real Map Integration**:
   * Integrate with Mapbox or Google Maps
   * Implement real-time location tracking
   * Add route visualization
3. **CRUD Operations**:
   * Complete forms for creating and editing shipments
   * Implement validation and error handling
   * Add confirmation dialogs for destructive actions

**Medium-term Goals**

1. **DeepCAL Calculator Implementation**:
   * Develop the Neutrosophic AHP-TOPSIS model
   * Create the decision engine UI
   * Implement dynamic weight sliders and ranking tables
2. **Advanced Analytics**:
   * Predictive analytics for risk assessment
   * Machine learning integration for route optimization
   * More comprehensive reporting tools
3. **Notification System**:
   * Real-time alerts for critical events
   * Email and push notification integration
   * Customizable alert thresholds

**Long-term Vision**

1. **AI Integration**:
   * Predictive risk modeling
   * Automated decision support
   * Natural language processing for commands
2. **Mobile Application**:
   * Native mobile apps for iOS and Android
   * Offline-first architecture
   * Location services integration
3. **IoT Integration**:
   * Connect with tracking devices
   * Sensor data integration
   * Automated status updates

**9. Testing and Deployment**

**Testing Strategy**

While the current implementation does not include comprehensive testing, the following testing strategy is recommended:

**Unit Testing**

* Test individual components in isolation
* Use Jest and React Testing Library
* Focus on component rendering and user interactions
* Mock Supabase client for data fetching tests

*// Example unit test for a component*

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

import { DashboardStats } from './dashboard-stats'

describe('DashboardStats', () => {

it('renders the correct number of stat cards', () => {

const stats = {

total\_shipments: 100,

active\_shipments: 50,

high\_risk\_shipments: 10,

delayed\_shipments: 5,

}

render(<DashboardStats stats={stats} />)

const cards = screen.getAllByRole('article')

expect(cards).toHaveLength(4)

})

})

**Integration Testing**

* Test interactions between components
* Focus on data flow and state management
* Use Cypress or Playwright for browser-based testing

*// Example Cypress integration test*

describe('Dashboard Page', () => {

beforeEach(() => {

cy.intercept('GET', '/api/stats', { fixture: 'stats.json' }).as('getStats')

cy.intercept('GET', '/api/shipments', { fixture: 'shipments.json' }).as('getShipments')

cy.intercept('GET', '/api/forwarders', { fixture: 'forwarders.json' }).as('getForwarders')

cy.visit('/')

cy.wait(['@getStats', '@getShipments', '@getForwarders'])

})

it('displays the correct number of stat cards', () => {

cy.get('[data-testid="stat-card"]').should('have.length', 4)

})

it('displays the shipment risk chart', () => {

cy.get('[data-testid="shipment-risk-chart"]').should('be.visible')

})

})

**End-to-End Testing**

* Test complete user flows
* Focus on critical paths and edge cases
* Use Cypress or Playwright for browser automation

*// Example E2E test*

describe('Shipment Details Flow', () => {

beforeEach(() => {

cy.login('test@example.com', 'password')

cy.visit('/shipments')

})

it('navigates to shipment details when clicking on a shipment', () => {

cy.get('[data-testid="shipment-row"]').first().click()

cy.url().should('include', '/shipments/')

cy.get('[data-testid="shipment-details"]').should('be.visible')

})

})

**Deployment**

**Development Environment**

* Local development using Next.js development server
* Supabase local development setup for database and authentication
* Environment variables for configuration

**Staging Environment**

* Vercel Preview Deployments for frontend
* Supabase staging project for backend
* Automated deployments from feature branches

**Production Environment**

* Vercel for frontend hosting
* Supabase for backend services
* Environment variables for configuration
* Continuous integration and deployment pipeline

**Deployment Process**

1. **Build Process**:
   * Run tests
   * Build Next.js application
   * Generate static assets
2. **Deployment**:
   * Deploy to Vercel
   * Update environment variables
   * Run database migrations (if needed)
3. **Post-Deployment**:
   * Run smoke tests
   * Monitor application performance
   * Check for errors in logs

**10. API Documentation**

**Supabase API Endpoints**

DeepCAL uses Supabase for data storage and retrieval. The following are the key API endpoints used in the application:

**Shipments**

**Get All Shipments**

const { data, error } = await supabase

.from("shipments")

.select(`

id,

request\_reference,

origin,

destination,

status,

risk\_level,

estimated\_arrival,

actual\_arrival,

priority,

created\_at,

forwarder\_id,

forwarders(name)

`)

.order("created\_at", { ascending: false })

* **Purpose**: Fetch all shipments with related forwarder information
* **Response**: Array of shipment objects with forwarder details
* **Error Handling**: Returns error object if the request fails

**Get Shipment by ID**

const { data, error } = await supabase

.from("shipments")

.select(`

id,

request\_reference,

origin,

destination,

status,

risk\_level,

estimated\_arrival,

actual\_arrival,

priority,

created\_at,

forwarder\_id,

customer\_id,

forwarders(

id,

name,

performance\_score

)

`)

.eq("id", id)

.single()

* **Purpose**: Fetch a specific shipment by ID with related forwarder information
* **Parameters**: id - The ID of the shipment to fetch
* **Response**: Shipment object with forwarder details
* **Error Handling**: Returns error object if the request fails or the shipment is not found

**Get Active Shipments**

const { data, error } = await supabase

.from("shipments")

.select("\*")

.eq("status", "in transit")

.order("risk\_level", { ascending: false })

.limit(10)

* **Purpose**: Fetch active shipments (in transit)
* **Response**: Array of active shipment objects
* **Error Handling**: Returns error object if the request fails

**Get Shipments with Location**

const { data, error } = await supabase

.from("shipments")

.select("\*")

.not("location", "is", null)

.order("risk\_level", { ascending: false })

* **Purpose**: Fetch shipments with location data for mapping
* **Response**: Array of shipment objects with location data
* **Error Handling**: Returns error object if the request fails

**Forwarders**

**Get All Forwarders**

const { data, error } = await supabase

.from("forwarders")

.select("\*")

.order("performance\_score", { ascending: false })

* **Purpose**: Fetch all forwarders ordered by performance score
* **Response**: Array of forwarder objects
* **Error Handling**: Returns error object if the request fails

**Get Top Performing Forwarders**

const { data, error } = await supabase

.from("forwarders")

.select("\*")

.order("performance\_score", { ascending: false })

.limit(5)

* **Purpose**: Fetch the top performing forwarders
* **Response**: Array of forwarder objects
* **Error Handling**: Returns error object if the request fails

**Real-Time Subscriptions**

**Shipment Location Updates**

const subscription = supabase

.channel("shipment-location-changes")

.on(

"postgres\_changes",

{

event: "UPDATE",

schema: "public",

table: "shipments",

filter: "location=neq.null",

},

(payload) => {

*// Handle update*

},

)

.subscribe()

* **Purpose**: Subscribe to updates to shipment locations
* **Events**: UPDATE
* **Filter**: Only shipments with non-null location
* **Callback**: Function to handle the update payload

**Shipment Status Changes**

const subscription = supabase

.channel("shipment-status-changes")

.on(

"postgres\_changes",

{

event: "\*",

schema: "public",

table: "shipments",

},

(payload) => {

*// Handle update*

},

)

.subscribe()

* **Purpose**: Subscribe to all changes to shipments
* **Events**: All (INSERT, UPDATE, DELETE)
* **Callback**: Function to handle the payload based on event type

**Specific Shipment Updates**

const subscription = supabase

.channel(`shipment-${id}`)

.on(

"postgres\_changes",

{

event: "UPDATE",

schema: "public",

table: "shipments",

filter: `id=eq.${id}`,

},

(payload) => {

*// Handle update*

},

)

.subscribe()

* **Purpose**: Subscribe to updates to a specific shipment
* **Parameters**: id - The ID of the shipment to monitor
* **Events**: UPDATE
* **Filter**: Only the specified shipment
* **Callback**: Function to handle the update payload

**Planned API Endpoints**

The following API endpoints are planned for future development:

**Authentication**

*// Sign in*

const { data, error } = await supabase.auth.signInWithPassword({

email,

password,

})

*// Sign out*

const { error } = await supabase.auth.signOut()

*// Get current user*

const { data: { user } } = await supabase.auth.getUser()

**DeepCAL Calculator**

*// Calculate optimal route*

const { data, error } = await fetch('/api/calculate-route', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({

origin,

destination,

priority,

weight,

volume,

}),

})

*// Calculate risk assessment*

const { data, error } = await fetch('/api/assess-risk', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({

shipmentId,

factors: {

weather: true,

political: true,

infrastructure: true,

},

}),

})

**Conclusion**

DeepCAL represents a sophisticated emergency logistics decision-support system with a modern, responsive interface and real-time capabilities. The application successfully implements the core features outlined in the requirements, with a solid foundation for future development.

The current implementation provides:

* Comprehensive shipment tracking and management
* Real-time data updates via Supabase
* Interactive data visualization
* Responsive, accessible UI with dark/light mode support
* Modular architecture for future expansion

The application successfully implements the SuperFuturistic UI/UX design approach with animated components, micro-interactions, and a clean, information-dense interface that prioritizes usability while maintaining visual appeal.

Future development will focus on implementing the DeepCAL Calculator with the Neutrosophic AHP-TOPSIS model, enhancing the analytics capabilities, and adding advanced features for emergency logistics management.

Based on the provided code and the current state of the DeepCAL application. Update app as shared below

DeepCAL: Intelligent Logistics and Analytics Dashboard Table of Contents Introduction

System Architecture

UI Pages & Component Breakdown

3.1 Mapview (Main Page)

3.2 Analytics (Shipment, Destination, Freight, Deep Insights)

3.3 DeepCAL Calculator

3.4 Settings

Data Integration & APIs

Database Schema

Security & Performance Considerations

Conclusion

1. Introduction DeepCAL is an AI-powered logistics and analytics platform optimized for emergency response and supply chain efficiency. The system integrates real-time shipment tracking, freight forwarder evaluation, and deep analytical insights, leveraging the Neutrosophic AHP-TOPSIS model for intelligent decision-making.

This documentation outlines the interface structure, components, and data flow of DeepCAL, ensuring a well-defined, scalable, and intuitive user experience.

1. System Architecture DeepCAL follows a modular design, consisting of:

Frontend (Next.js, Tailwind CSS, shadcn/ui) for an interactive user interface.

Backend (FastAPI, Python-based decision engine) for complex analytics and AI computations.

Database (Supabase - PostgreSQL) for data storage, real-time updates, and API services.

Edge Functions (Supabase serverless logic) for background processing and automation.

Each module interacts seamlessly through well-defined REST APIs.

1. UI Pages & Component Breakdown 3.1 Mapview (Main Page) Description A full-screen interactive map for shipment tracking, geospatial risk assessment, and logistics optimization.

Key Features Live Shipment Tracking: Displays real-time shipment locations using Mapbox GL.

Geospatial Risk Indicators: Color-coded overlays for high-risk areas based on real-time data.

Interactive Filtering: Users can toggle shipment categories (e.g., high-priority, delayed, at-risk).

Route Optimization: AI-powered recommendations for alternate routing based on risk analytics.

Component Breakdown Component Functionality Map Widget Displays real-time locations, routes, and heatmaps. Risk Heatmap Highlights high-risk zones for shipments. Filter Controls Allow users to refine displayed shipments by priority, status, etc. Detailed Shipment View Clickable markers provide shipment details, including risk scores. 3.2 Analytics Page Description A dashboard containing multiple analytic insights into shipment performance, risk factors, and freight efficiency.

Tabs & Insights Tab Features Shipment Analytics Graphs and tables on delivery times, cost efficiency, and real-time shipment data. Destination Analytics Evaluates risk levels and delivery success rates across destinations. Freight Analytics Performance and reliability scores of freight forwarders. Deep Insights AI-driven predictions on shipment risks, cost variations, and supply chain bottlenecks. Component Breakdown Component Functionality Bar & Line Charts Displays historical and predictive shipment trends. Performance Scorecards Summarizes key logistics metrics (OTIF %, delay rates, cost deviation). Interactive Data Tables Allows filtering and deep dives into logistics performance. Heatmap View Geospatial visualization of high-risk destinations. 3.3 DeepCAL Calculator Description A real-time logistics decision engine that computes the best freight options, shipment routes, and risk adjustments.

Key Features Neutrosophic AHP-TOPSIS Computation: Uses AI-based ranking models to evaluate shipment alternatives.

Automated Risk Assessment: Dynamically adjusts weightings based on historical and real-time factors.

Cost-Performance Optimization: Recommends the most efficient routes and freight forwarders.

Component Breakdown Component Functionality Decision Engine UI Allows users to input shipment criteria and run calculations. Dynamic Weight Sliders Adjusts factor importance (cost, reliability, responsiveness). Ranking Table Displays optimal options ranked based on AHP-TOPSIS. AI Insights Provides reasoning and recommendations for selected shipment plans. 3.4 Settings Description A configuration panel for managing DeepCAL system preferences, user roles, API integrations, and visual themes.

Key Features General System Settings: Manage analytics update frequency, alert thresholds, and API integrations.

User Role Management: Define access control levels for different stakeholders.

Customization & Themes: Adapt UI themes (light/dark mode), adjust dashboard layout.

API & Integration Management: Configure connections with Supabase, Mapbox, OpenWeather API, etc.

1. Data Integration & APIs DeepCAL integrates with multiple external services and APIs for real-time insights.

API Function Supabase Database API Handles storage of shipment, freight, and risk data. Mapbox API Provides real-time mapping, geospatial risk visualization. OpenWeather API Supplies weather conditions for risk assessment. DeepCAL Decision Engine (FastAPI) Runs AHP-TOPSIS ranking and risk analysis. 5. Database Schema DeepCAL utilizes a normalized PostgreSQL schema in Supabase.

Shipment Table sql Copy Edit CREATE TABLE shipments ( id SERIAL PRIMARY KEY, origin VARCHAR(255), destination VARCHAR(255), weight FLOAT, volume FLOAT, cost FLOAT, status VARCHAR(50), estimated\_arrival TIMESTAMP ); Freight Forwarders Table sql Copy Edit CREATE TABLE freight\_forwarders ( id SERIAL PRIMARY KEY, name VARCHAR(255), service\_type VARCHAR(255), cost\_metric FLOAT, reliability\_score FLOAT ); Risk Assessment Table sql Copy Edit CREATE TABLE risk\_assessment ( id SERIAL PRIMARY KEY, shipment\_id INT REFERENCES shipments(id), risk\_score FLOAT, delay\_probability FLOAT, reason TEXT ); API Request Logging sql Copy Edit CREATE TABLE api\_logs ( id SERIAL PRIMARY KEY, endpoint VARCHAR(255), request\_time TIMESTAMP, response\_time FLOAT, status\_code INT ); 6. Security & Performance Considerations Authentication: Uses Supabase Auth (JWT-based).

Rate Limiting: Prevents excessive API calls.

Caching Strategies: Edge functions optimize data retrieval to improve responsiveness.

Monitoring & Logging: Logs API calls and performance metrics for system health tracking.

1. Conclusion DeepCAL is an advanced AI-powered logistics analytics and decision-making tool, integrating real-time tracking, predictive risk assessment, and automated freight optimization. Its architecture supports scalability, security, and seamless integration with modern APIs and decision-making models.

This document serves as the foundation for accurate implementation and development, ensuring that DeepCAL provides an intuitive and data-driven interface for efficient emergency logistics and supply chain manage