🎯 CURSOR BACKEND MASTER PROMPT

Football Jackpot Probability Engine - Complete Backend Implementation

Target: Production-ready backend using Supabase + Edge Functions + PostgreSQL
Complexity: Senior Backend Engineer + Quantitative Analyst Level
Output: Fully functional API matching frontend contract

📋 ROLE & CONTEXT

You are a **Principal Backend Engineer** + **Sports Analytics Quant** building a professional probability estimation system for football jackpot betting. This is NOT a simple CRUD API - it's a statistically rigorous modeling platform.

Core Requirements:

- ✅ Statistical rigor (Dixon-Coles Poisson model)
- ✅ Production-grade code (typed, tested, documented)
- ✅ Supabase-native architecture (Edge Functions + PostgreSQL)
- ✅ Complete API contract matching frontend
- ✅ NO neural networks, NO black boxes

🏗️ ARCHITECTURE OVERVIEW

Technology Stack (MANDATORY)

Platform: Supabase (PostgreSQL 15 + Edge Functions)
Runtime: Deno (for Edge Functions)
Language: TypeScript (strict mode)
Database: PostgreSQL with Row Level Security
Validation: Zod schemas
Math Library: Custom Dixon-Coles implementation
Data Sources: Football-Data.co.uk (CSV) + API-Football (optional)

Why Supabase?

- ✅ Built-in PostgreSQL (no separate database hosting)
- ✅ Edge Functions (serverless, globally distributed)
- ✅ Row Level Security (built-in user isolation)
- ✅ Real-time subscriptions (for background tasks)
- ✅ Auto-generated REST API (for simple CRUD)
- ✅ File storage (for model weights, CSV uploads)

📁 PROJECT STRUCTURE

```
supabase/
├── config.toml                 # Supabase configuration
├── seed.sql                    # Initial data (leagues, teams)
│
├── migrations/                 # Database schema migrations
│   ├── 20250101000000_initial_schema.sql
│   ├── 20250102000000_add_indexes.sql
│   └── 20250103000000_enable_rls.sql
│
└── functions/                  # Edge Functions (API endpoints)
    │
    ├── _shared/                # Shared utilities (import via relative paths)
    │   ├── cors.ts             # CORS headers for all functions
    │   ├── validation.ts       # Zod schemas for input validation
    │   ├── db.ts               # Database connection helper
    │   ├── errors.ts           # Custom error classes
    │   ├── dixon-coles.ts      # Dixon-Coles Poisson model
    │   ├── calibration.ts      # Isotonic regression
    │   ├── probability-sets.ts # 7 probability set generators
    │   └── types.ts            # TypeScript type definitions
    │
    ├── calculate-probabilities/  # POST /api/v1/predictions
    │   ├── index.ts            # Main handler
    │   └── README.md           # Endpoint documentation
    │
    ├── get-prediction/         # GET /api/v1/predictions/:id
    │   ├── index.ts
```

```
│   └── README.md
│
├── model-status/              # GET /api/v1/model/status
│   ├── index.ts
│   └── README.md
│
├── model-health/              # GET /api/v1/health/model
│   ├── index.ts
│   └── README.md
│
├── validation-metrics/        # GET /api/v1/validation/metrics
│   ├── index.ts
│   └── README.md
│
├── data-refresh/              # POST /api/v1/data/refresh
│   ├── index.ts
│   └── README.md
│
├── train-model/               # POST /api/v1/model/train
│   ├── index.ts
│   └── README.md
│
├── task-status/               # GET /api/v1/tasks/:taskId
│   ├── index.ts
│   └── README.md
│
├── export-csv/                # GET /api/v1/predictions/:id/export
│   ├── index.ts
│   └── README.md
│
└── team-search/               # GET /api/v1/teams/search
    ├── index.ts
    └── README.md
```

🗄️ DATABASE SCHEMA

CRITICAL: Use the exact schema provided in the Lovable document. I'll provide the complete implementation-ready SQL.

Core Tables (Must Implement)

1. **leagues** - Reference data for football leagues
2. **teams** - Team registry with attack/defense ratings
3. **matches** - Historical match results (training data)
4. **team_features** - Rolling statistics (feature store)
5. **league_stats** - League-level aggregates
6. **models** - Model version registry
7. **training_runs** - Training history
8. **jackpots** - User jackpot inputs
9. **jackpot_fixtures** - Individual fixtures within jackpots
10. **predictions** - Calculated probabilities (7 sets per fixture)
11. **validation_results** - Prediction accuracy tracking
12. **calibration_data** - For isotonic regression
13. **data_sources** - Data ingestion tracking
14. **ingestion_logs** - ETL job logs

Key Design Decisions

Enums:

```sql
CREATE TYPE match_outcome AS ENUM ('H', 'D', 'A');
CREATE TYPE model_status AS ENUM ('training', 'active', 'archived', 'failed');
CREATE TYPE probability_set_type AS ENUM (
    'pure_model',       -- Set A
    'market_aware',     -- Set B
    'conservative',     -- Set C
    'draw_boosted',     -- Set D
    'sharp',            -- Set E
    'kelly',            -- Set F
    'ensemble'          -- Set G
);
```

Row Level Security:

- Users see only their own jackpots
- Public read for leagues/teams/models
- Admin-only write for models/training

🎯 CORE MODELING LOGIC

1. Dixon-Coles Implementation (`_shared/dixon-coles.ts`)

Mathematical Foundation:

```
typescript
```

```typescript
/**
 * Dixon-Coles Poisson Model for Football Match Prediction
 *
 * Reference: "Modelling Association Football Scores and Inefficiencies
 * in the Football Betting Market" (Dixon & Coles, 1997)
 *
 * Key Concepts:
 * - Attack/Defense strength parameters per team
 * - Home advantage factor
 * - Low-score dependency parameter (ρ)
 * - Exponential time decay (ξ)
 */

interface TeamStrength {
  teamId: string;
  attack: number;    // α_i
  defense: number;   // β_i
  homeAdvantage?: number;
}

interface DixonColesParams {
  rho: number;          // Dependency parameter (typically -0.13)
  xi: number;           // Time decay (0.0065 per day)
  homeAdvantage: number; // Global home advantage (0.3-0.5 goals)
}

interface GoalExpectations {
  lambdaHome: number;    // Expected home goals
  lambdaAway: number;    // Expected away goals
}

interface MatchProbabilities {
  home: number;  // P(Home Win)
  draw: number;  // P(Draw)
  away: number;  // P(Away Win)
  entropy: number;
}

/**
 * Calculate expected goals for a match
 *
 * λ_home = exp(α_home - β_away + γ)
 * λ_away = exp(α_away - β_home)
```

```
 *
 * where γ is home advantage
 */
function calculateExpectedGoals(
  homeTeam: TeamStrength,
  awayTeam: TeamStrength,
  params: DixonColesParams
): GoalExpectations {
  const lambdaHome = Math.exp(
    homeTeam.attack - awayTeam.defense + params.homeAdvantage
  );

  const lambdaAway = Math.exp(
    awayTeam.attack - homeTeam.defense
  );

  return { lambdaHome, lambdaAway };
}

/**
 * Poisson probability: P(X = k) = (λ^k * e^(-λ)) / k!
 */
function poissonProbability(lambda: number, k: number): number {
  return (Math.pow(lambda, k) * Math.exp(-lambda)) / factorial(k);
}

/**
 * Dixon-Coles adjustment factor for low scores
 *
 * τ(x, y, λ_home, λ_away, ρ) adjusts probabilities for:
 * - (0,0), (1,0), (0,1), (1,1)
 */
function tauAdjustment(
  homeGoals: number,
  awayGoals: number,
  lambdaHome: number,
  lambdaAway: number,
  rho: number
): number {
  // Only adjust for low-score combinations
  if (homeGoals > 1 || awayGoals > 1) return 1.0;

  if (homeGoals === 0 && awayGoals === 0) {
    return 1 - lambdaHome * lambdaAway * rho;
```

```
  }
  if (homeGoals === 0 && awayGoals === 1) {
    return 1 + lambdaHome * rho;
  }
  if (homeGoals === 1 && awayGoals === 0) {
    return 1 + lambdaAway * rho;
  }
  if (homeGoals === 1 && awayGoals === 1) {
    return 1 - rho;
  }

  return 1.0;
}

/**
 * Calculate joint probability of a score
 *
 * P(X=x, Y=y) = τ(x,y) * Poisson(x; λ_home) * Poisson(y; λ_away)
 */
function scoreJointProbability(
  homeGoals: number,
  awayGoals: number,
  lambdaHome: number,
  lambdaAway: number,
  rho: number
): number {
  const poissonHome = poissonProbability(lambdaHome, homeGoals);
  const poissonAway = poissonProbability(lambdaAway, awayGoals);
  const tau = tauAdjustment(homeGoals, awayGoals, lambdaHome, lambdaAway, rho);

  return tau * poissonHome * poissonAway;
}

/**
 * Calculate match outcome probabilities
 *
 * P(Home) = Σ P(x, y) for x > y
 * P(Draw) = Σ P(x, x) for all x
 * P(Away) = Σ P(x, y) for x < y
 */
function calculateMatchProbabilities(
  homeTeam: TeamStrength,
  awayTeam: TeamStrength,
  params: DixonColesParams,
```

```
  maxGoals: number = 8
): MatchProbabilities {
  const { lambdaHome, lambdaAway } = calculateExpectedGoals(
    homeTeam,
    awayTeam,
    params
  );

  let probHome = 0;
  let probDraw = 0;
  let probAway = 0;

  // Sum over all possible scores up to maxGoals
  for (let h = 0; h <= maxGoals; h++) {
    for (let a = 0; a <= maxGoals; a++) {
      const prob = scoreJointProbability(h, a, lambdaHome, lambdaAway, params.rho);

      if (h > a) probHome += prob;
      else if (h === a) probDraw += prob;
      else probAway += prob;
    }
  }

  // Normalize (should sum to ~1.0 already)
  const total = probHome + probDraw + probAway;
  probHome /= total;
  probDraw /= total;
  probAway /= total;

  // Calculate entropy
  const entropy = -[probHome, probDraw, probAway].reduce((sum, p) => {
    return p > 0 ? sum + p * Math.log2(p) : sum;
  }, 0);

  return {
    home: probHome,
    draw: probDraw,
    away: probAway,
    entropy
  };
}

/**
 * Train team strength parameters from historical matches
```

```
 *
 * Uses Maximum Likelihood Estimation with exponential time decay
 */
async function trainTeamStrengths(
  matches: HistoricalMatch[],
  params: DixonColesParams
): Promise<Map<string, TeamStrength>> {
  // Implementation details:
  // 1. Initialize all teams with α=1.0, β=1.0
  // 2. Apply exponential weights based on match age
  // 3. Use Newton-Raphson or gradient descent for MLE
  // 4. Regularize to prevent overfitting (L2 penalty)
  // 5. Return team strength map

  // This is computationally intensive - implement iteratively
  // See: https://github.com/dashee87/blogScripts/blob/master/R/2017-06-04-predicting-football-results-with-statistical-model

  throw new Error("Implement MLE training - see architecture doc Section D");
}
```

2. Market Odds Blending (`_shared/blending.ts`)

typescript

```typescript
/**
 * Blend model probabilities with market-implied probabilities
 *
 * Uses learned weights from historical validation data
 */

interface MarketOdds {
  home: number;
  draw: number;
  away: number;
}

interface BlendingWeights {
  modelWeight: number;    // α (0.0 to 1.0)
  marketWeight: number;   // 1 - α
}

/**
 * Convert odds to implied probabilities (remove bookmaker margin)
 */
function oddsToImpliedProbabilities(odds: MarketOdds): MatchProbabilities {
  const rawProbs = {
    home: 1 / odds.home,
    draw: 1 / odds.draw,
    away: 1 / odds.away
  };

  // Normalize to remove margin
  const total = rawProbs.home + rawProbs.draw + rawProbs.away;

  return {
    home: rawProbs.home / total,
    draw: rawProbs.draw / total,
    away: rawProbs.away / total,
    entropy: 0 // Calculate after blending
  };
}

/**
 * Blend model and market probabilities
 *
 * P_blended = α * P_model + (1 - α) * P_market
 */
```

```
function blendProbabilities(
  modelProbs: MatchProbabilities,
  marketOdds: MarketOdds,
  weights: BlendingWeights
): MatchProbabilities {
  const marketProbs = oddsToImpliedProbabilities(marketOdds);

  const blended = {
    home: weights.modelWeight * modelProbs.home + weights.marketWeight * marketProbs.home,
    draw: weights.modelWeight * modelProbs.draw + weights.marketWeight * marketProbs.draw,
    away: weights.modelWeight * modelProbs.away + weights.marketWeight * marketProbs.away,
    entropy: 0
  };

  // Recalculate entropy
  blended.entropy = -[blended.home, blended.draw, blended.away].reduce(
    (sum, p) => p > 0 ? sum + p * Math.log2(p) : sum,
    0
  );

  return blended;
}

/**
 * Learn optimal blending weights from validation data
 *
 * Minimize Brier score on historical predictions vs actuals
 */
async function learnBlendingWeights(
  validationData: ValidationMatch[]
): Promise<BlendingWeights> {
  // Grid search over α ∈ [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
  // For each α:
  //   - Blend all validation matches
  //   - Calculate Brier score
  // Return α with lowest Brier score

  throw new Error("Implement blending weight optimization");
}
```

3. Isotonic Calibration (`_shared/calibration.ts`)

typescript

```typescript
/**
 * Isotonic Regression for Probability Calibration
 *
 * Maps predicted probabilities → calibrated probabilities
 * to ensure P(outcome | prediction=p) ≈ p
 */

interface CalibrationCurve {
  outcome: 'H' | 'D' | 'A';
  predictedBuckets: number[];    // [0.0, 0.1, 0.2, ..., 1.0]
  observedFrequencies: number[]; // Actual frequencies in each bucket
}

/**
 * Fit isotonic regression from historical data
 */
function fitIsotonicRegression(
  predictions: number[],
  actuals: number[]  // 0 or 1 (outcome occurred or not)
): CalibrationCurve {
  // 1. Sort predictions and actuals together
  // 2. Group into buckets (e.g., 0.0-0.1, 0.1-0.2, ...)
  // 3. Calculate observed frequency in each bucket
  // 4. Apply isotonic constraint (non-decreasing)

  throw new Error("Implement isotonic regression fitting");
}

/**
 * Apply calibration to a new prediction
 */
function calibrateProbability(
  rawProbability: number,
  calibrationCurve: CalibrationCurve
): number {
  // Linear interpolation between calibrated buckets

  const { predictedBuckets, observedFrequencies } = calibrationCurve;

  // Find surrounding buckets
  let lowerIdx = 0;
  for (let i = 0; i < predictedBuckets.length - 1; i++) {
    if (rawProbability >= predictedBuckets[i] &&
```

```
      rawProbability < predictedBuckets[i + 1]) {
    lowerIdx = i;
    break;
   }
  }

  const lowerBucket = predictedBuckets[lowerIdx];
  const upperBucket = predictedBuckets[lowerIdx + 1] || 1.0;
  const lowerFreq = observedFrequencies[lowerIdx];
  const upperFreq = observedFrequencies[lowerIdx + 1] || observedFrequencies[lowerIdx];

  // Linear interpolation
  const t = (rawProbability - lowerBucket) / (upperBucket - lowerBucket);
  return lowerFreq + t * (upperFreq - lowerFreq);
}
```

4. Probability Set Generators (`_shared/probability-sets.ts`)

typescript

```
/**
 * Generate all 7 probability sets from base calculations
 */

interface ProbabilitySetConfig {
  id: 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G';
  name: string;
  modelWeight: number;
  marketWeight: number;
  drawBoost?: number;
  temperatureAdjustment?: number;
}

const PROBABILITY_SET_CONFIGS: ProbabilitySetConfig[] = [
  {
    id: 'A',
    name: 'Pure Model',
    modelWeight: 1.0,
    marketWeight: 0.0
  },
  {
    id: 'B',
    name: 'Balanced',
    modelWeight: 0.6,
    marketWeight: 0.4
  },
  {
    id: 'C',
    name: 'Market-Dominant',
    modelWeight: 0.2,
    marketWeight: 0.8
  },
  {
    id: 'D',
    name: 'Draw-Boosted',
    modelWeight: 0.6,
    marketWeight: 0.4,
    drawBoost: 1.15
  },
  {
    id: 'E',
    name: 'Entropy-Penalized',
    modelWeight: 0.6,
```

```
    marketWeight: 0.4,
    temperatureAdjustment: 1.5
  },
  {
    id: 'F',
    name: 'Kelly-Weighted',
    modelWeight: 0.6,
    marketWeight: 0.4
    // Special handling: weight by Kelly criterion
  },
  {
    id: 'G',
    name: 'Ensemble',
    modelWeight: 0.0,  // Computed as average of A, B, C
    marketWeight: 0.0
  }
];

/**
 * Generate all probability sets for a fixture
 */
function generateAllProbabilitySets(
  modelProbs: MatchProbabilities,
  marketOdds: MarketOdds,
  calibrationCurves: Map<string, CalibrationCurve>
): Record<string, MatchProbabilities> {
  const sets: Record<string, MatchProbabilities> = {};

  for (const config of PROBABILITY_SET_CONFIGS) {
    let probs: MatchProbabilities;

    if (config.id === 'G') {
      // Ensemble: average of A, B, C
      probs = ensembleProbabilities([sets['A'], sets['B'], sets['C']]);
    } else {
      // Blend model and market
      probs = blendProbabilities(
        modelProbs,
        marketOdds,
        { modelWeight: config.modelWeight, marketWeight: config.marketWeight }
      );

      // Apply draw boost if configured
      if (config.drawBoost) {
```

```
      probs = boostDrawProbability(probs, config.drawBoost);
    }

    // Apply temperature adjustment if configured
    if (config.temperatureAdjustment) {
      probs = adjustTemperature(probs, config.temperatureAdjustment);
    }

    // Calibrate
    probs = calibrateProbabilities(probs, calibrationCurves);
  }

  sets[config.id] = probs;
 }

 return sets;
}
```

🔌 API ENDPOINTS IMPLEMENTATION

Endpoint 1: Calculate Probabilities

Path: `supabase/functions/calculate-probabilities/index.ts`

```
typescript
```

```
import { serve } from "https://deno.land/std@0.168.0/http/server.ts";
import { createClient } from "https://esm.sh/@supabase/supabase-js@2";
import { z } from "https://deno.land/x/zod@v3.22.4/mod.ts";
import { corsHeaders } from "../_shared/cors.ts";
import { calculateMatchProbabilities } from "../_shared/dixon-coles.ts";
import { generateAllProbabilitySets } from "../_shared/probability-sets.ts";

// Request schema
const CalculateProbabilitiesSchema = z.object({
  fixtures: z.array(z.object({
    id: z.string(),
    homeTeam: z.string(),
    awayTeam: z.string(),
    odds: z.object({
      home: z.number().min(1.01).max(100),
      draw: z.number().min(1.01).max(100),
      away: z.number().min(1.01).max(100)
    }).nullable()
  })),
  createdAt: z.string().datetime()
});

serve(async (req) => {
  // Handle CORS
  if (req.method === "OPTIONS") {
    return new Response(null, { headers: corsHeaders });
  }

  try {
    // Parse and validate request
    const body = await req.json();
    const { fixtures, createdAt } = CalculateProbabilitiesSchema.parse(body);

    // Initialize Supabase client
    const supabaseClient = createClient(
      Deno.env.get("SUPABASE_URL") ?? "",
      Deno.env.get("SUPABASE_ANON_KEY") ?? ""
    );

    // Get active model
    const { data: model } = await supabaseClient
      .from("models")
      .select("*")
```

```
  .eq("status", "active")
  .single();

if (!model) {
  throw new Error("No active model found");
}

// Create jackpot entry
const { data: jackpot } = await supabaseClient
  .from("jackpots")
  .insert({
    jackpot_id: `JK-${Date.now()}`,
    user_id: req.headers.get("user-id"), // From auth
    status: "calculating"
  })
  .select()
  .single();

// Process each fixture
const predictions = [];

for (const fixture of fixtures) {
  // 1. Resolve team names to IDs
  const { homeTeamId, awayTeamId, leagueId } = await resolveTeams(
    supabaseClient,
    fixture.homeTeam,
    fixture.awayTeam
  );

  // 2. Get team strengths
  const homeStrength = await getTeamStrength(supabaseClient, homeTeamId);
  const awayStrength = await getTeamStrength(supabaseClient, awayTeamId);

  // 3. Calculate model probabilities
  const modelProbs = calculateMatchProbabilities(
    homeStrength,
    awayStrength,
    model.model_weights.params
  );

  // 4. Generate all probability sets
  const probabilitySets = generateAllProbabilitySets(
    modelProbs,
    fixture.odds,
```

```
    model.model_weights.calibration_curves
  );

  // 5. Store fixture and predictions
  const { data: fixtureRecord } = await supabaseClient
    .from("jackpot_fixtures")
    .insert({
      jackpot_id: jackpot.id,
      match_order: fixtures.indexOf(fixture) + 1,
      home_team: fixture.homeTeam,
      away_team: fixture.awayTeam,
      odds_home: fixture.odds?.home,
      odds_draw: fixture.odds?.draw,
      odds_away: fixture.odds?.away,
      home_team_id: homeTeamId,
      away_team_id: awayTeamId,
      league_id: leagueId
    })
    .select()
    .single();

  // 6. Store all probability sets
  for (const [setId, probs] of Object.entries(probabilitySets)) {
    await supabaseClient
      .from("predictions")
      .insert({
        fixture_id: fixtureRecord.id,
        model_id: model.id,
        set_type: setIdToDbEnum(setId),
        prob_home: probs.home,
        prob_draw: probs.draw,
        prob_away: probs.away,
        predicted_outcome: maxProbabilityOutcome(probs),
        confidence: Math.max(probs.home, probs.draw, probs.away),
        expected_home_goals: modelProbs.lambdaHome,
        expected_away_goals: modelProbs.lambdaAway
      });
  }

  predictions.push({
    fixture: fixture,
    probabilitySets: probabilitySets
  });
}
```

```
    // Update jackpot status
    await supabaseClient
      .from("jackpots")
      .update({ status: "calculated" })
      .eq("id", jackpot.id);

    // Return response matching frontend contract
    return new Response(
      JSON.stringify({
        predictionId: jackpot.id,
        modelVersion: model.version,
        createdAt: new Date().toISOString(),
        fixtures: fixtures,
        probabilitySets: transformToProbabilitySets(predictions),
        confidenceFlags: generateConfidenceFlags(predictions)
      }),
      {
        headers: { ...corsHeaders, "Content-Type": "application/json" },
        status: 200
      }
    );

  } catch (error) {
    console.error("Error:", error);
    return new Response(
      JSON.stringify({ error: error.message }),
      {
        headers: { ...corsHeaders, "Content-Type": "application/json" },
        status: 400
      }
    );
  }
});
```

Endpoint 2: Model Training

Path: supabase/functions/train-model/index.ts

typescript

```typescript
import { serve } from "https://deno.land/std@0.168.0/http/server.ts";
import { corsHeaders } from "../_shared/cors.ts";
import { trainTeamStrengths } from "../_shared/dixon-coles.ts";

serve(async (req) => {
  if (req.method === "OPTIONS") {
    return new Response(null, { headers: corsHeaders });
  }

  try {
    // This is a long-running operation
    // Return immediately with task ID, process asynchronously

    const taskId = crypto.randomUUID();

    // Queue training job (use Supabase Functions queue or pg_cron)
    // For now, return task ID and process synchronously (timeout: 60s)

    const supabaseClient = createClient(
      Deno.env.get("SUPABASE_URL") ?? "",
      Deno.env.get("SUPABASE_ANON_KEY") ?? ""
    );

    // 1. Load historical matches
    const { data: matches } = await supabaseClient
      .from("matches")
      .select("*")
      .order("match_date", { ascending: false })
      .limit(5000);

    // 2. Train Dixon-Coles parameters
    const teamStrengths = await trainTeamStrengths(
      matches,
      {
        rho: -0.13,
        xi: 0.0065,
        homeAdvantage: 0.35
      }
    );

    // 3. Learn blending weights
    // (Split data into train/validation)
```

```
    // 4. Fit calibration curves

    // 5. Save model
    const { data: model } = await supabaseClient
      .from("models")
      .insert({
        version: `v${Date.now()}`,
        model_type: "dixon-coles",
        status: "active",
        training_completed_at: new Date().toISOString(),
        model_weights: {
          team_strengths: Array.from(teamStrengths.entries()),
          params: { rho: -0.13, xi: 0.0065, homeAdvantage: 0.35 },
          calibration_curves: {}  // Add fitted curves
        },
        brier_score: 0.187  // Calculate from validation
      })
      .select()
      .single();

    return new Response(
      JSON.stringify({ taskId, modelId: model.id }),
      {
        headers: { ...corsHeaders, "Content-Type": "application/json" },
        status: 202
      }
    );

  } catch (error) {
    return new Response(
      JSON.stringify({ error: error.message }),
      {
        headers: { ...corsHeaders, "Content-Type": "application/json" },
        status: 500
      }
    );
  }
});
```

📝 IMPLEMENTATION CHECKLIST

Phase 1: Database Setup (Week 1)

- [ ] Create Supabase project
- [ ] Run initial schema migration
- [ ] Seed leagues and teams data
- [ ] Enable Row Level Security
- [ ] Add indexes for performance
- [ ] Test CRUD operations via Supabase Studio

Phase 2: Core Math Implementation (Week 1-2)

- [ ] Implement Dixon-Coles in `_shared/dixon-coles.ts`
- [ ] Implement isotonic calibration in `_shared/calibration.ts`
- [ ] Implement probability set generators
- [ ] Write unit tests for all math functions
- [ ] Validate against known results

Phase 3: API Endpoints (Week 2-3)

- [ ] Implement `/calculate-probabilities`
- [ ] Implement `/get-prediction`
- [ ] Implement `/model-status`
- [ ] Implement `/model-health`
- [ ] Implement `/validation-metrics`
- [ ] Implement `/data-refresh`
- [ ] Implement `/train-model`
- [ ] Implement `/task-status`

Phase 4: Data Ingestion (Week 3)

- [ ] CSV parser for Football-Data.co.uk
- [ ] Team name normalization
- [ ] Duplicate detection
- [ ] Validation and error handling
- [ ] Automated daily downloads

Phase 5: Testing & Validation (Week 4)

- [ ] Unit tests for all functions

- [ ] Integration tests for API endpoints
- [ ] Load testing (1000+ concurrent requests)
- [ ] Validate Brier scores < 0.20
- [ ] Test all probability sets
- [ ] Frontend-backend integration testing

🎯 CRITICAL SUCCESS CRITERIA

Before considering the backend "production-ready":

1. ✅ All API endpoints return correct TypeScript types
2. ✅ Dixon-Coles implementation validated against academic papers
3. ✅ Brier score < 0.20 on validation set
4. ✅ All 7 probability sets calibrated independently
5. ✅ Reliability curves approximately diagonal
6. ✅ Response time < 500ms for single fixture
7. ✅ Response time < 3s for 13-fixture jackpot
8. ✅ Database queries optimized with proper indexes
9. ✅ Row Level Security prevents data leakage
10. ✅ Error handling covers all edge cases

🚀 DEPLOYMENT INSTRUCTIONS

Local Development

```bash
```

```bash
# Install Supabase CLI
npm install -g supabase

# Initialize project
supabase init

# Start local Supabase stack
supabase start

# Run migrations
supabase db reset

# Deploy functions locally
supabase functions serve

# Test endpoint
curl -X POST http://localhost:54321/functions/v1/calculate-probabilities \
  -H "Content-Type: application/json" \
  -d '{"fixtures": [...]}'
```

Production Deployment

```bash
# Link to Supabase project
supabase link --project-ref YOUR_PROJECT_REF

# Push database schema
supabase db push

# Deploy Edge Functions
supabase functions deploy calculate-probabilities
supabase functions deploy model-status
# ... deploy all functions

# Set environment variables
supabase secrets set API_FOOTBALL_KEY=your_key_here
```

📚 ADDITIONAL RESOURCES

Mathematical References

- Dixon & Coles (1997): Original paper
- Rue & Salvesen (2000): Extensions to Dixon-Coles
- Scikit-learn Isotonic Regression docs

Code References

- https://github.com/dashee87/blogScripts (R implementation)
- https://github.com/martineastwood/penaltyblog (Python implementation)

Data Sources

- Football-Data.co.uk: CSV download scripts
- API-Football: API documentation

⚠️ CRITICAL REMINDERS

1. **NO Neural Networks**: This is classical statistics, stick to Dixon-Coles
2. **Type Safety**: Use Zod for all inputs, strict TypeScript everywhere
3. **Calibration is Mandatory**: Isotonic regression is NOT optional
4. **Test with Real Data**: Minimum 5 seasons per league
5. **Performance Matters**: Cache team strengths, optimize queries
6. **Security First**: Row Level Security, input validation, rate limiting
7. **Documentation**: Every function needs docstrings with examples

---

🎓 LEARNING PATH (If Implementing from Scratch)

Day 1-2: Understand Dixon-Coles

- Read the original 1997 paper

- Implement basic Poisson model first
- Add Dixon-Coles dependency parameter
- Validate against known results

Day 3-4: Database & API Structure

- Set up Supabase project
- Create schema
- Implement first endpoint (calculate-probabilities)
- Test with sample data

Day 5-7: Complete Implementation

- Implement all endpoints
- Add calibration
- Generate all probability sets
- Integration testing

Day 8-10: Optimization & Deployment

- Performance tuning
- Load testing
- Deploy to Supabase
- Connect frontend

This prompt contains everything needed to build a production-ready backend. Follow the structure exactly, implement the math rigorously, and you'll have a professional probability engine that matches the frontend perfectly.

Good luck, and remember: statistical rigor over complexity.