# **AI Insights Pipeline Implementation Plan**

Based on the ProgressPlay Reporting Platform documentation, I'll provide a comprehensive implementation for an AI-powered insights pipeline that surfaces actionable analytics directly within the user interface.

## **Architecture Overview**

┌────────────────┐ ┌─────────────────┐ ┌────────────────┐ ┌─────────────────┐

│ Data Sources │────▶│ Analysis Engine │────▶│ Insights Store │────▶│ Delivery System │

└────────────────┘ └─────────────────┘ └────────────────┘ └─────────────────┘

│ ▲ │ │

│ │ │ │

│ ┌─────────────┐ │ │

└───────────────│ ML Models │◀──────────────┘ │

└─────────────┘ │

│

┌────────────────────────────────────────────────────┘

│

▼

┌────────────────────────────────────────────────────────────┐

│ User Interface │

│ ┌────────────────┐ ┌────────────────┐ ┌────────────┐ │

│ │ Insight Cards │ │ Trend Spotters │ │ Anomalies │ │

│ └────────────────┘ └────────────────┘ └────────────┘ │

└────────────────────────────────────────────────────────────┘

## **1. Backend Implementation (C#)**

### **Core Services**

// 1. Insight Service - Core Service

public class InsightService : IInsightService

{

private readonly IReportService \_reportService;

private readonly IAnalyticsEngine \_analyticsEngine;

private readonly IInsightRepository \_insightRepository;

private readonly IHubContext<NotificationHub> \_hubContext;

private readonly ILogger<InsightService> \_logger;

public InsightService(

IReportService reportService,

IAnalyticsEngine analyticsEngine,

IInsightRepository insightRepository,

IHubContext<NotificationHub> hubContext,

ILogger<InsightService> logger)

{

\_reportService = reportService;

\_analyticsEngine = analyticsEngine;

\_insightRepository = insightRepository;

\_hubContext = hubContext;

\_logger = logger;

}

public async Task<IEnumerable<Insight>> GetInsightsForDashboard(string dashboardId, string userId)

{

// Get user role and permissions

var userContext = await \_userService.GetUserContext(userId);

// Fetch recent insights from repository with proper filtering

var insights = await \_insightRepository.GetInsightsForDashboard(

dashboardId,

userContext.Role,

userContext.Permissions);

return insights;

}

public async Task GenerateInsights(InsightGenerationRequest request)

{

try

{

// Fetch data for analysis

var reportData = await \_reportService.GetReportData(request.ReportId, request.Parameters);

// Process data through analytics engine

var insights = await \_analyticsEngine.AnalyzeData(reportData, request.AnalysisTypes);

// Store insights

await \_insightRepository.StoreInsights(insights);

// Notify relevant users via SignalR

await NotifyUsers(insights);

\_logger.LogInformation($"Generated {insights.Count()} insights for report {request.ReportId}");

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Error generating insights for report {request.ReportId}");

throw;

}

}

private async Task NotifyUsers(IEnumerable<Insight> insights)

{

foreach (var insight in insights)

{

var targetUsers = await \_userService.GetUsersForWhitelabel(insight.WhitelabelId);

foreach (var user in targetUsers)

{

await \_hubContext.Clients.User(user.Id).SendAsync(

"InsightGenerated",

new InsightNotification(insight.Id, insight.Title, insight.Severity));

}

}

}

}

// 2. Analytics Engine - Core ML Implementation

public class AnalyticsEngine : IAnalyticsEngine

{

private readonly IEnumerable<IInsightAnalyzer> \_analyzers;

private readonly ILogger<AnalyticsEngine> \_logger;

private readonly IMLModelClient \_mlModelClient;

public AnalyticsEngine(

IEnumerable<IInsightAnalyzer> analyzers,

IMLModelClient mlModelClient,

ILogger<AnalyticsEngine> logger)

{

\_analyzers = analyzers;

\_mlModelClient = mlModelClient;

\_logger = logger;

}

public async Task<IEnumerable<Insight>> AnalyzeData(

ReportData data,

IEnumerable<AnalysisType> analysisTypes)

{

var insights = new List<Insight>();

// Run appropriate analyzers based on requested analysis types

foreach (var type in analysisTypes)

{

var analyzersForType = \_analyzers.Where(a => a.SupportsAnalysisType(type));

foreach (var analyzer in analyzersForType)

{

try

{

var analyzerInsights = await analyzer.Analyze(data);

insights.AddRange(analyzerInsights);

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Error running analyzer {analyzer.GetType().Name}");

}

}

}

// Enrichment with ML predictions when appropriate

if (analysisTypes.Contains(AnalysisType.Prediction))

{

try

{

var predictions = await \_mlModelClient.GetPredictions(data);

insights.AddRange(

predictions.Select(p => new Insight

{

Title = p.Title,

Description = p.Description,

Category = InsightCategory.Prediction,

Severity = p.Confidence > 0.8 ? InsightSeverity.High : InsightSeverity.Medium,

RelatedMetrics = p.RelatedMetrics,

WhitelabelId = data.WhitelabelId,

Confidence = p.Confidence,

GeneratedAt = DateTime.UtcNow

}));

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error fetching ML predictions");

}

}

return insights;

}

}

// 3. Insight Analyzers - Specialized analysis components

public class TrendAnalyzer : IInsightAnalyzer

{

public bool SupportsAnalysisType(AnalysisType type) =>

type == AnalysisType.Trend || type == AnalysisType.All;

public async Task<IEnumerable<Insight>> Analyze(ReportData data)

{

var insights = new List<Insight>();

// Identify significant trends in time series data

foreach (var metric in data.Metrics.Where(m => m.DataType == DataType.Numeric))

{

var timeSeriesData = data.GetTimeSeriesData(metric.Name);

if (timeSeriesData.Count() < 5) continue; // Need sufficient data points

// Calculate moving averages

var movingAvg = CalculateMovingAverage(timeSeriesData, 3);

// Detect consistent upward or downward trends

var trend = DetectTrend(movingAvg);

if (trend.Significance > 0.7)

{

insights.Add(new Insight

{

Title = $"{metric.DisplayName} {(trend.Direction == TrendDirection.Up ? "increasing" : "decreasing")}",

Description = $"{metric.DisplayName} has {trend.Direction.ToString().ToLower()}ed by {trend.PercentageChange:P} " +

$"over the last {timeSeriesData.Count()} periods.",

Category = InsightCategory.Trend,

Severity = trend.Significance > 0.9 ? InsightSeverity.High : InsightSeverity.Medium,

RelatedMetrics = new[] { metric.Name },

WhitelabelId = data.WhitelabelId,

Confidence = trend.Significance,

GeneratedAt = DateTime.UtcNow

});

}

}

return insights;

}

private IEnumerable<double> CalculateMovingAverage(IEnumerable<double> values, int window)

{

// Implementation of moving average calculation

var result = new List<double>();

var valuesList = values.ToList();

for (int i = 0; i <= valuesList.Count - window; i++)

{

var avg = valuesList.Skip(i).Take(window).Average();

result.Add(avg);

}

return result;

}

private TrendResult DetectTrend(IEnumerable<double> values)

{

// Trend detection logic using linear regression

// Returns direction, significance and percentage change

// Simple implementation for demonstration

var valuesList = values.ToList();

var firstValue = valuesList.First();

var lastValue = valuesList.Last();

if (Math.Abs(firstValue) < 0.0001) return new TrendResult(TrendDirection.Neutral, 0, 0);

var percentChange = (lastValue - firstValue) / Math.Abs(firstValue);

var direction = percentChange > 0 ? TrendDirection.Up :

percentChange < 0 ? TrendDirection.Down : TrendDirection.Neutral;

// Calculate R-squared or other significance metric

var significance = CalculateSignificance(valuesList, direction);

return new TrendResult(direction, significance, Math.Abs(percentChange));

}

private double CalculateSignificance(List<double> values, TrendDirection direction)

{

// Calculate statistical significance of trend

// Implementation depends on statistical method chosen

// Simplified version for demo

var n = values.Count;

if (n < 3) return 0;

// Check consistency of direction

var consistentPoints = 0;

for (int i = 1; i < n; i++)

{

var diff = values[i] - values[i-1];

if ((direction == TrendDirection.Up && diff > 0) ||

(direction == TrendDirection.Down && diff < 0))

{

consistentPoints++;

}

}

return (double)consistentPoints / (n - 1);

}

}

public class AnomalyDetector : IInsightAnalyzer

{

public bool SupportsAnalysisType(AnalysisType type) =>

type == AnalysisType.Anomaly || type == AnalysisType.All;

public async Task<IEnumerable<Insight>> Analyze(ReportData data)

{

var insights = new List<Insight>();

// Detect anomalies using statistical methods

foreach (var metric in data.Metrics.Where(m => m.DataType == DataType.Numeric))

{

var timeSeriesData = data.GetTimeSeriesData(metric.Name);

if (timeSeriesData.Count() < 7) continue;

// Apply Z-score or other anomaly detection method

var anomalies = DetectAnomalies(timeSeriesData);

foreach (var anomaly in anomalies)

{

insights.Add(new Insight

{

Title = $"Unusual {metric.DisplayName} detected",

Description = $"{metric.DisplayName} showed an unusual value of {anomaly.Value:N2} " +

$"on {anomaly.Date.ToShortDateString()}, which is {anomaly.DeviationPercentage:P} " +

$"{(anomaly.Direction == AnomalyDirection.Above ? "above" : "below")} expected range.",

Category = InsightCategory.Anomaly,

Severity = anomaly.Severity,

RelatedMetrics = new[] { metric.Name },

WhitelabelId = data.WhitelabelId,

Confidence = anomaly.Confidence,

RelatedDate = anomaly.Date,

GeneratedAt = DateTime.UtcNow

});

}

}

return insights;

}

private IEnumerable<AnomalyResult> DetectAnomalies(IEnumerable<TimeSeriesDataPoint> dataPoints)

{

var values = dataPoints.Select(d => d.Value).ToList();

var dates = dataPoints.Select(d => d.Date).ToList();

// Calculate mean and standard deviation

var mean = values.Average();

var stdDev = CalculateStandardDeviation(values, mean);

var results = new List<AnomalyResult>();

// Z-score threshold for anomalies

const double threshold = 2.5;

for (int i = 0; i < values.Count; i++)

{

var value = values[i];

var zScore = Math.Abs(value - mean) / stdDev;

if (zScore > threshold)

{

var direction = value > mean ? AnomalyDirection.Above : AnomalyDirection.Below;

var deviation = Math.Abs(value - mean) / mean;

var confidence = Math.Min(1.0, (zScore - threshold) / 2 + 0.7);

var severity = zScore > 3.5 ? InsightSeverity.High :

zScore > 3.0 ? InsightSeverity.Medium : InsightSeverity.Low;

results.Add(new AnomalyResult

{

Date = dates[i],

Value = value,

ZScore = zScore,

Direction = direction,

DeviationPercentage = deviation,

Confidence = confidence,

Severity = severity

});

}

}

return results;

}

private double CalculateStandardDeviation(List<double> values, double mean)

{

if (values.Count <= 1) return 0;

var sumOfSquares = values.Sum(v => Math.Pow(v - mean, 2));

return Math.Sqrt(sumOfSquares / (values.Count - 1));

}

}

// 4. Background Service - Scheduled insight generation

public class InsightGenerationService : BackgroundService

{

private readonly IServiceProvider \_serviceProvider;

private readonly ILogger<InsightGenerationService> \_logger;

private readonly IOptions<InsightGenerationOptions> \_options;

public InsightGenerationService(

IServiceProvider serviceProvider,

IOptions<InsightGenerationOptions> options,

ILogger<InsightGenerationService> logger)

{

\_serviceProvider = serviceProvider;

\_options = options;

\_logger = logger;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

\_logger.LogInformation("Insight Generation Service is starting");

while (!stoppingToken.IsCancellationRequested)

{

try

{

await GenerateScheduledInsights();

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occurred while generating insights");

}

// Wait for the next scheduled run

await Task.Delay(\_options.Value.ScheduleIntervalMinutes \* 60 \* 1000, stoppingToken);

}

}

private async Task GenerateScheduledInsights()

{

using var scope = \_serviceProvider.CreateScope();

var reportService = scope.ServiceProvider.GetRequiredService<IReportService>();

var insightService = scope.ServiceProvider.GetRequiredService<IInsightService>();

// Get list of dashboards to analyze

var dashboards = await reportService.GetActiveDashboards();

foreach (var dashboard in dashboards)

{

\_logger.LogInformation($"Generating insights for dashboard {dashboard.Id}");

var request = new InsightGenerationRequest

{

ReportId = dashboard.DefaultReportId,

Parameters = dashboard.DefaultParameters,

AnalysisTypes = new[]

{

AnalysisType.Trend,

AnalysisType.Anomaly,

AnalysisType.Comparison,

AnalysisType.Prediction

}

};

await insightService.GenerateInsights(request);

}

}

}

// 5. ML Model Client - Integration with ML services

public class MLModelClient : IMLModelClient

{

private readonly HttpClient \_httpClient;

private readonly IOptions<MLModelOptions> \_options;

private readonly ILogger<MLModelClient> \_logger;

public MLModelClient(

HttpClient httpClient,

IOptions<MLModelOptions> options,

ILogger<MLModelClient> logger)

{

\_httpClient = httpClient;

\_options = options;

\_logger = logger;

}

public async Task<IEnumerable<PredictionResult>> GetPredictions(ReportData data)

{

try

{

var request = new MLPredictionRequest

{

WhitelabelId = data.WhitelabelId,

TimeSeriesData = data.TimeSeriesData,

MetricDefinitions = data.Metrics,

PredictionWindow = \_options.Value.DefaultPredictionWindow

};

var response = await \_httpClient.PostAsJsonAsync(

\_options.Value.PredictionEndpoint,

request);

response.EnsureSuccessStatusCode();

var result = await response.Content.ReadFromJsonAsync<MLPredictionResponse>();

return result.Predictions;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error getting predictions from ML service");

throw;

}

}

}

### **Data Models**

// Insight Data Model

public class Insight

{

public string Id { get; set; } = Guid.NewGuid().ToString();

public string Title { get; set; }

public string Description { get; set; }

public InsightCategory Category { get; set; }

public InsightSeverity Severity { get; set; }

public string[] RelatedMetrics { get; set; }

public string WhitelabelId { get; set; }

public double Confidence { get; set; }

public DateTime? RelatedDate { get; set; }

public DateTime GeneratedAt { get; set; }

public bool IsRead { get; set; }

public string[] Tags { get; set; }

public Dictionary<string, object> AdditionalData { get; set; }

}

public enum InsightCategory

{

Trend,

Anomaly,

Comparison,

Correlation,

Prediction,

Recommendation

}

public enum InsightSeverity

{

Low,

Medium,

High,

Critical

}

public enum AnalysisType

{

Trend,

Anomaly,

Comparison,

Correlation,

Prediction,

All

}

// Request and Response Models

public class InsightGenerationRequest

{

public string ReportId { get; set; }

public Dictionary<string, object> Parameters { get; set; }

public IEnumerable<AnalysisType> AnalysisTypes { get; set; }

}

public class InsightNotification

{

public string InsightId { get; set; }

public string Title { get; set; }

public InsightSeverity Severity { get; set; }

public InsightNotification(string insightId, string title, InsightSeverity severity)

{

InsightId = insightId;

Title = title;

Severity = severity;

}

}

// Analysis Result Models

public class TrendResult

{

public TrendDirection Direction { get; }

public double Significance { get; }

public double PercentageChange { get; }

public TrendResult(TrendDirection direction, double significance, double percentageChange)

{

Direction = direction;

Significance = significance;

PercentageChange = percentageChange;

}

}

public enum TrendDirection

{

Up,

Down,

Neutral

}

public class AnomalyResult

{

public DateTime Date { get; set; }

public double Value { get; set; }

public double ZScore { get; set; }

public AnomalyDirection Direction { get; set; }

public double DeviationPercentage { get; set; }

public double Confidence { get; set; }

public InsightSeverity Severity { get; set; }

}

public enum AnomalyDirection

{

Above,

Below

}

// ML Integration Models

public class MLPredictionRequest

{

public string WhitelabelId { get; set; }

public Dictionary<string, IEnumerable<TimeSeriesDataPoint>> TimeSeriesData { get; set; }

public IEnumerable<MetricDefinition> MetricDefinitions { get; set; }

public int PredictionWindow { get; set; }

}

public class MLPredictionResponse

{

public IEnumerable<PredictionResult> Predictions { get; set; }

}

public class PredictionResult

{

public string Title { get; set; }

public string Description { get; set; }

public string[] RelatedMetrics { get; set; }

public double Confidence { get; set; }

public Dictionary<string, double> PredictedValues { get; set; }

}

public class TimeSeriesDataPoint

{

public DateTime Date { get; set; }

public double Value { get; set; }

}

public class MetricDefinition

{

public string Name { get; set; }

public string DisplayName { get; set; }

public DataType DataType { get; set; }

public string Format { get; set; }

}

public enum DataType

{

Numeric,

Text,

Boolean,

Date

}

### **API Controllers**

[ApiController]

[Route("api/[controller]")]

[Authorize]

public class InsightsController : ControllerBase

{

private readonly IInsightService \_insightService;

private readonly ILogger<InsightsController> \_logger;

public InsightsController(

IInsightService insightService,

ILogger<InsightsController> logger)

{

\_insightService = insightService;

\_logger = logger;

}

[HttpGet("dashboard/{dashboardId}")]

public async Task<IActionResult> GetInsightsForDashboard(string dashboardId)

{

try

{

var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

var insights = await \_insightService.GetInsightsForDashboard(dashboardId, userId);

return Ok(insights);

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Error getting insights for dashboard {dashboardId}");

return StatusCode(500, "An error occurred while retrieving insights");

}

}

[HttpPost("generate")]

[Authorize(Roles = "Admin")]

public async Task<IActionResult> GenerateInsights([FromBody] InsightGenerationRequest request)

{

try

{

await \_insightService.GenerateInsights(request);

return Ok(new { message = "Insight generation started successfully" });

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error starting insight generation");

return StatusCode(500, "An error occurred while starting insight generation");

}

}

[HttpPut("{insightId}/read")]

public async Task<IActionResult> MarkAsRead(string insightId)

{

try

{

var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

await \_insightService.MarkInsightAsRead(insightId, userId);

return Ok();

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Error marking insight {insightId} as read");

return StatusCode(500, "An error occurred while updating the insight");

}

}

}

### **Dependency Registration**

// In Program.cs or Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// Register insight services

services.AddScoped<IInsightService, InsightService>();

services.AddScoped<IInsightRepository, InsightRepository>();

services.AddScoped<IAnalyticsEngine, AnalyticsEngine>();

// Register analyzers

services.AddScoped<IInsightAnalyzer, TrendAnalyzer>();

services.AddScoped<IInsightAnalyzer, AnomalyDetector>();

services.AddScoped<IInsightAnalyzer, ComparisonAnalyzer>();

services.AddScoped<IInsightAnalyzer, CorrelationAnalyzer>();

// Configure ML client

services.AddHttpClient<IMLModelClient, MLModelClient>(client =>

{

client.BaseAddress = new Uri(Configuration["MLService:BaseUrl"]);

client.DefaultRequestHeaders.Add("x-api-key", Configuration["MLService:ApiKey"]);

});

// Configure options

services.Configure<MLModelOptions>(Configuration.GetSection("MLService"));

services.Configure<InsightGenerationOptions>(Configuration.GetSection("InsightGeneration"));

// Register background service

services.AddHostedService<InsightGenerationService>();

}

## **2. Frontend Implementation (React)**

### **Core Components**

// 1. InsightsPanel.tsx - Main container for insights

import React, { useEffect, useState } from 'react';

import { useSelector, useDispatch } from 'react-redux';

import { fetchInsights, markInsightAsRead } from '../store/insightsSlice';

import InsightCard from './InsightCard';

import InsightFilter from './InsightFilter';

import { Insight, InsightCategory, InsightSeverity } from '../types/insights';

import { RootState } from '../store/store';

interface InsightsPanelProps {

dashboardId: string;

onInsightSelected?: (insight: Insight) => void;

}

const InsightsPanel: React.FC<InsightsPanelProps> = ({

dashboardId,

onInsightSelected

}) => {

const dispatch = useDispatch();

const { insights, loading, error } = useSelector(

(state: RootState) => state.insights

);

const [filters, setFilters] = useState({

categories: [] as InsightCategory[],

severity: [] as InsightSeverity[],

onlyUnread: false

});

useEffect(() => {

dispatch(fetchInsights(dashboardId));

}, [dispatch, dashboardId]);

const handleFilterChange = (newFilters: typeof filters) => {

setFilters(newFilters);

};

const handleInsightClick = (insight: Insight) => {

if (!insight.isRead) {

dispatch(markInsightAsRead(insight.id));

}

if (onInsightSelected) {

onInsightSelected(insight);

}

};

const filteredInsights = insights.filter(insight => {

// Apply category filter

if (filters.categories.length > 0 &&

!filters.categories.includes(insight.category)) {

return false;

}

// Apply severity filter

if (filters.severity.length > 0 &&

!filters.severity.includes(insight.severity)) {

return false;

}

// Apply unread filter

if (filters.onlyUnread && insight.isRead) {

return false;

}

return true;

});

return (

<div className="insights-panel">

<div className="insights-panel-header">

<h3>Insights</h3>

<InsightFilter

filters={filters}

onChange={handleFilterChange}

/>

</div>

{loading ? (

<div className="insights-loading">Loading insights...</div>

) : error ? (

<div className="insights-error">

Error loading insights: {error}

</div>

) : filteredInsights.length === 0 ? (

<div className="insights-empty">

No insights match your current filters

</div>

) : (

<div className="insights-list">

{filteredInsights.map(insight => (

<InsightCard

key={insight.id}

insight={insight}

onClick={() => handleInsightClick(insight)}

/>

))}

</div>

)}

</div>

);

};

export default InsightsPanel;

// 2. InsightCard.tsx - Individual insight display

import React from 'react';

import { Insight, InsightCategory, InsightSeverity } from '../types/insights';

import { format } from 'date-fns';

import {

TrendingUp,

TrendingDown,

AlertTriangle,

BarChart2,

Activity,

Award

} from 'lucide-react';

interface InsightCardProps {

insight: Insight;

onClick?: () => void;

}

const InsightCard: React.FC<InsightCardProps> = ({ insight, onClick }) => {

const getIcon = () => {

switch (insight.category) {

case InsightCategory.Trend:

return insight.additionalData?.direction === 'Up'

? <TrendingUp />

: <TrendingDown />;

case InsightCategory.Anomaly:

return <AlertTriangle />;

case InsightCategory.Comparison:

return <BarChart2 />;

case InsightCategory.Correlation:

return <Activity />;

case InsightCategory.Prediction:

case InsightCategory.Recommendation:

return <Award />;

default:

return <AlertTriangle />;

}

};

const getSeverityClass = () => {

switch (insight.severity) {

case InsightSeverity.Critical:

return 'critical';

case InsightSeverity.High:

return 'high';

case InsightSeverity.Medium:

return 'medium';

case InsightSeverity.Low:

return 'low';

default:

return '';

}

};

return (

<div

className={`insight-card ${getSeverityClass()} ${insight.isRead ? 'read' : 'unread'}`}

onClick={onClick}

>

<div className="insight-card-icon">

{getIcon()}

</div>

<div className="insight-card-content">

<h4 className="insight-card-title">{insight.title}</h4>

<p className="insight-card-description">{insight.description}</p>

<div className="insight-card-meta">

{insight.relatedDate && (

<span className="insight-date">

{format(new Date(insight.relatedDate), 'MMM d, yyyy')}

</span>

)}

<span className="insight-confidence">

{Math.round(insight.confidence \* 100)}% confidence

</span>

</div>

</div>

</div>

);

};

export default InsightCard;

// 3. InsightDetail.tsx - Expanded insight view with actions

import React from 'react';

import { Insight, InsightCategory } from '../types/insights';

import { format } from 'date-fns';

import InsightActions from './InsightActions';

import InsightMetricChart from './InsightMetricChart';

interface InsightDetailProps {

insight: Insight;

onClose: () => void;

}

const InsightDetail: React.FC<InsightDetailProps> = ({ insight, onClose }) => {

return (

<div className="insight-detail">

<div className="insight-detail-header">

<h3>{insight.title}</h3>

<button className="close-button" onClick={onClose}>×</button>

</div>

<div className="insight-detail-content">

<p className="insight-description">{insight.description}</p>

{insight.relatedMetrics && insight.relatedMetrics.length > 0 && (

<div className="insight-metrics">

<h4>Related Metrics</h4>

{insight.relatedMetrics.map(metric => (

<InsightMetricChart

key={metric}

metricName={metric}

insightId={insight.id}

category={insight.category}

relatedDate={insight.relatedDate}

/>

))}

</div>

)}

{insight.category === InsightCategory.Prediction && (

<div className="insight-prediction">

<h4>Prediction Details</h4>

<div className="prediction-values">

{Object.entries(insight.additionalData?.predictedValues || {}).map(([key, value]) => (

<div key={key} className="prediction-value">

<span className="metric-name">{key}:</span>

<span className="metric-value">{value}</span>

</div>

))}

</div>

</div>

)}

<div className="insight-meta">

<div className="meta-item">

<span className="meta-label">Generated:</span>

<span className="meta-value">

{format(new Date(insight.generatedAt), 'MMM d, yyyy h:mm a')}

</span>

</div>

<div className="meta-item">

<span className="meta-label">Confidence:</span>

<span className="meta-value">

{Math.round(insight.confidence \* 100)}%

</span>

</div>

{insight.tags && insight.tags.length > 0 && (

<div className="meta-item">

<span className="meta-label">Tags:</span>

<div className="meta-tags">

{insight.tags.map(tag => (

<span key={tag} className="meta-tag">{tag}</span>

))}

</div>

</div>

)}

</div>

</div>

<InsightActions insight={insight} />

</div>

);

};

export default InsightDetail;

// 4. InsightMetricChart.tsx - Chart visualization for insight metrics

import React, { useEffect, useState } from 'react';

import { useSelector } from 'react-redux';

import { RootState } from '../store/store';

import { InsightCategory } from '../types/insights';

import {

LineChart,

Line,

XAxis,

YAxis,

CartesianGrid,

Tooltip,

ResponsiveContainer,

ReferenceLine,

ReferenceArea

} from 'recharts';

import { fetchMetricData } from '../api/metricsApi';

interface InsightMetricChartProps {

metricName: string;

insightId: string;

category: InsightCategory;

relatedDate?: Date;

}

const InsightMetricChart: React.FC<InsightMetricChartProps> = ({

metricName,

insightId,

category,

relatedDate

}) => {

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true);

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

const metricDefinition = useSelector(

(state: RootState) => state.metrics.definitions.find(m => m.name === metricName)

);

useEffect(() => {

const loadData = async () => {

try {

setLoading(true);

const result = await fetchMetricData(metricName, insightId);

setData(result);

setError(null);

} catch (err) {

setError('Failed to load metric data');

console.error(err);

} finally {

setLoading(false);

}

};

loadData();

}, [metricName, insightId]);

if (loading) {

return <div className="metric-loading">Loading chart data...</div>;

}

if (error) {

return <div className="metric-error">{error}</div>;

}

// Format tooltip value based on metric type

const formatValue = (value: number) => {

if (!metricDefinition || !metricDefinition.format) {

return value;

}

switch (metricDefinition.format) {

case 'currency':

return new Intl.NumberFormat('en-US', {

style: 'currency',

currency: 'USD'

}).format(value);

case 'percentage':

return new Intl.NumberFormat('en-US', {

style: 'percent',

minimumFractionDigits: 2

}).format(value / 100);

default:

return value;

}

};

// Create visualization based on insight category

const renderChart = () => {

const commonProps = {

data,

margin: { top: 5, right: 20, bottom: 5, left: 0 }

};

switch (category) {

case InsightCategory.Anomaly:

return (

<ResponsiveContainer width="100%" height={200}>

<LineChart {...commonProps}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="date" />

<YAxis />

<Tooltip formatter={formatValue} />

<Line

type="monotone"

dataKey="value"

stroke="#8884d8"

activeDot={{ r: 8 }}

/>

{relatedDate && (

<ReferenceLine

x={relatedDate.toISOString().split('T')[0]}

stroke="red"

strokeWidth={2}

label="Anomaly"

/>

)}

</LineChart>

</ResponsiveContainer>

);

case InsightCategory.Trend:

// Find trend start and end points

const trendStart = data.length > 0 ? data[0].date : null;

const trendEnd = data.length > 0 ? data[data.length - 1].date : null;

return (

<ResponsiveContainer width="100%" height={200}>

<LineChart {...commonProps}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="date" />

<YAxis />

<Tooltip formatter={formatValue} />

<Line

type="monotone"

dataKey="value"

stroke="#8884d8"

activeDot={{ r: 8 }}

/>

{trendStart && trendEnd && (

<ReferenceArea

x1={trendStart}

x2={trendEnd}

strokeOpacity={0.3}

fill="#8884d8"

fillOpacity={0.1}

/>

)}

</LineChart>

</ResponsiveContainer>

);

default:

return (

<ResponsiveContainer width="100%" height={200}>

<LineChart {...commonProps}>

<CartesianGrid strokeDasharray="3 3" />

<XAxis dataKey="date" />

<YAxis />

<Tooltip formatter={formatValue} />

<Line

type="monotone"

dataKey="value"

stroke="#8884d8"

activeDot={{ r: 8 }}

/>

</LineChart>

</ResponsiveContainer>

);

}

};

return (

<div className="insight-metric-chart">

<h5>{metricDefinition?.displayName || metricName}</h5>

{renderChart()}

</div>

);

};

export default InsightMetricChart;

### **Redux Integration**

// insightsSlice.ts

import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

import { Insight } from '../types/insights';

import \* as insightsApi from '../api/insightsApi';

interface InsightsState {

insights: Insight[];

loading: boolean;

error: string | null;

}

const initialState: InsightsState = {

insights: [],

loading: false,

error: null

};

export const fetchInsights = createAsyncThunk(

'insights/fetchInsights',

async (dashboardId: string) => {

const response = await insightsApi.getInsightsForDashboard(dashboardId);

return response;

}

);

export const markInsightAsRead = createAsyncThunk(

'insights/markAsRead',

async (insightId: string) => {

await insightsApi.markInsightAsRead(insightId);

return insightId;

}

);

const insightsSlice = createSlice({

name: 'insights',

initialState,

reducers: {

addRealTimeInsight: (state, action) => {

// Add a new insight that came from SignalR

state.insights.unshift(action.payload);

}

},

extraReducers: (builder) => {

builder

.addCase(fetchInsights.pending, (state) => {

state.loading = true;

state.error = null;

})

.addCase(fetchInsights.fulfilled, (state, action) => {

state.loading = false;

state.insights = action.payload;

})

.addCase(fetchInsights.rejected, (state, action) => {

state.loading = false;

state.error = action.error.message || 'Failed to load insights';

})

.addCase(markInsightAsRead.fulfilled, (state, action) => {

const insight = state.insights.find(i => i.id === action.payload);

if (insight) {

insight.isRead = true;

}

});

}

});

export const { addRealTimeInsight } = insightsSlice.actions;

export default insightsSlice.reducer;

### **SignalR Integration**

// SignalRService.ts - Real-time notification integration

import { HubConnectionBuilder, HubConnection, LogLevel } from '@microsoft/signalr';

import { store } from '../store/store';

import { addRealTimeInsight } from '../store/insightsSlice';

import { Insight } from '../types/insights';

export class SignalRService {

private connection: HubConnection | null = null;

private static instance: SignalRService;

private constructor() {}

public static getInstance(): SignalRService {

if (!SignalRService.instance) {

SignalRService.instance = new SignalRService();

}

return SignalRService.instance;

}

public async startConnection(token: string) {

if (this.connection) {

return;

}

this.connection = new HubConnectionBuilder()

.withUrl('/hubs/notifications', {

accessTokenFactory: () => token

})

.withAutomaticReconnect()

.configureLogging(LogLevel.Information)

.build();

this.registerHandlers();

try {

await this.connection.start();

console.log('SignalR connection established');

} catch (err) {

console.error('Error establishing SignalR connection:', err);

}

}

public async stopConnection() {

if (this.connection) {

try {

await this.connection.stop();

this.connection = null;

console.log('SignalR connection stopped');

} catch (err) {

console.error('Error stopping SignalR connection:', err);

}

}

}

private registerHandlers() {

if (!this.connection) {

return;

}

this.connection.on('InsightGenerated', (notification: {

insightId: string;

title: string;

severity: number;

}) => {

// Fetch the full insight details

fetch(`/api/insights/${notification.insightId}`)

.then(response => response.json())

.then((insight: Insight) => {

store.dispatch(addRealTimeInsight(insight));

// Show browser notification if permitted

this.showBrowserNotification(insight);

})

.catch(err => console.error('Error fetching insight details:', err));

});

}

private showBrowserNotification(insight: Insight) {

if ('Notification' in window && Notification.permission === 'granted') {

new Notification('New Insight', {

body: insight.title,

icon: '/icons/insight\_icon.png'

});

}

}

}

export default SignalRService.getInstance();

### **Dashboard Integration**

// DashboardPage.tsx - Integration with existing dashboard

import React, { useState } from 'react';

import { useParams } from 'react-router-dom';

import DashboardHeader from './DashboardHeader';

import KpiGrid from './KpiGrid';

import ChartSection from './ChartSection';

import InsightsPanel from './insights/InsightsPanel';

import InsightDetail from './insights/InsightDetail';

import { Insight } from '../types/insights';

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

const { dashboardId } = useParams<{ dashboardId: string }>();

const [selectedInsight, setSelectedInsight] = useState<Insight | null>(null);

const [insightsPanelOpen, setInsightsPanelOpen] = useState(false);

const handleInsightSelected = (insight: Insight) => {

setSelectedInsight(insight);

};

const handleCloseInsightDetail = () => {

setSelectedInsight(null);

};

const toggleInsightsPanel = () => {

setInsightsPanelOpen(!insightsPanelOpen);

};

return (

<div className="dashboard-page">

<DashboardHeader

dashboardId={dashboardId}

onToggleInsights={toggleInsightsPanel}

insightCount={5} // This would come from redux state

/>

<div className="dashboard-content">

<div className="dashboard-main">

<KpiGrid dashboardId={dashboardId} />

<ChartSection dashboardId={dashboardId} />

</div>

{insightsPanelOpen && (

<div className="dashboard-insights-panel">

<InsightsPanel

dashboardId={dashboardId}

onInsightSelected={handleInsightSelected}

/>

</div>

)}

</div>

{selectedInsight && (

<div className="insight-detail-modal">

<InsightDetail

insight={selectedInsight}

onClose={handleCloseInsightDetail}

/>

</div>

)}

</div>

);

};

export default DashboardPage;

## **3. Database Integration**

-- Create insights table

CREATE TABLE dbo.Insights (

Id NVARCHAR(50) PRIMARY KEY,

Title NVARCHAR(200) NOT NULL,

Description NVARCHAR(1000) NOT NULL,

Category INT NOT NULL,

Severity INT NOT NULL,

WhitelabelId NVARCHAR(50) NOT NULL,

Confidence FLOAT NOT NULL,

RelatedDate DATETIME2 NULL,

GeneratedAt DATETIME2 NOT NULL,

RelatedMetrics NVARCHAR(MAX) NULL,

Tags NVARCHAR(MAX) NULL,

AdditionalData NVARCHAR(MAX) NULL

);

-- Create user insights read status table

CREATE TABLE dbo.UserInsights (

UserId NVARCHAR(50) NOT NULL,

InsightId NVARCHAR(50) NOT NULL,

IsRead BIT NOT NULL DEFAULT 0,

ReadAt DATETIME2 NULL,

PRIMARY KEY (UserId, InsightId),

FOREIGN KEY (InsightId) REFERENCES dbo.Insights(Id) ON DELETE CASCADE

);

-- Create index for faster querying

CREATE INDEX IX\_Insights\_WhitelabelId ON dbo.Insights(WhitelabelId);

CREATE INDEX IX\_Insights\_Category ON dbo.Insights(Category);

CREATE INDEX IX\_Insights\_GeneratedAt ON dbo.Insights(GeneratedAt);

-- Create stored procedure for retrieving insights

CREATE PROCEDURE dbo.GetInsightsForDashboard

@DashboardId NVARCHAR(50),

@UserId NVARCHAR(50),

@WhitelabelIds NVARCHAR(MAX),

@MaxResults INT = 100

AS

BEGIN

SET NOCOUNT ON;

-- Parse comma-separated whitelabel IDs into table

DECLARE @WhitelabelTable TABLE (WhitelabelId NVARCHAR(50));

INSERT INTO @WhitelabelTable

SELECT value FROM STRING\_SPLIT(@WhitelabelIds, ',');

-- Get insights with read status

SELECT TOP (@MaxResults)

i.Id,

i.Title,

i.Description,

i.Category,

i.Severity,

i.WhitelabelId,

i.Confidence,

i.RelatedDate,

i.GeneratedAt,

i.RelatedMetrics,

i.Tags,

i.AdditionalData,

ISNULL(ui.IsRead, 0) AS IsRead

FROM

dbo.Insights i

LEFT JOIN

dbo.UserInsights ui ON i.Id = ui.InsightId AND ui.UserId = @UserId

WHERE

i.WhitelabelId IN (SELECT WhitelabelId FROM @WhitelabelTable)

ORDER BY

i.GeneratedAt DESC;

END;

## **4. Performance Considerations**

1. **Caching Strategy**:  
   * Cache insight results in Redis with appropriate TTL
   * Pre-compute common insights during off-peak hours
   * Implement client-side caching with React Query
2. **Computation Optimization**:  
   * Use simplified algorithms for real-time analysis
   * Schedule complex computations as background tasks
   * Implement early termination for analyzers when confidence is low
3. **Database Optimization**:  
   * Create indexes for frequently queried fields
   * Implement time-based partitioning for historical insights
   * Use JSON data type for additional data to avoid schema changes
4. **Front-end Performance**:  
   * Lazy load insight details and charts
   * Implement virtual scrolling for long insight lists
   * Use web workers for client-side data processing

## **5. Integration Testing**

// Sample integration test for insight generation

[TestClass]

public class InsightGenerationTests

{

private Mock<IReportService> \_mockReportService;

private Mock<IInsightRepository> \_mockRepository;

private Mock<IAnalyticsEngine> \_mockAnalyticsEngine;

private Mock<IHubContext<NotificationHub>> \_mockHubContext;

private IInsightService \_insightService;

[TestInitialize]

public void Setup()

{

\_mockReportService = new Mock<IReportService>();

\_mockRepository = new Mock<IInsightRepository>();

\_mockAnalyticsEngine = new Mock<IAnalyticsEngine>();

\_mockHubContext = new Mock<IHubContext<NotificationHub>>();

var logger = new Mock<ILogger<InsightService>>();

\_insightService = new InsightService(

\_mockReportService.Object,

\_mockAnalyticsEngine.Object,

\_mockRepository.Object,

\_mockHubContext.Object,

logger.Object);

}

[TestMethod]

public async Task GenerateInsights\_CallsAnalyticsEngine\_AndStoresResults()

{

// Arrange

var request = new InsightGenerationRequest

{

ReportId = "test-report",

Parameters = new Dictionary<string, object>(),

AnalysisTypes = new[] { AnalysisType.Trend, AnalysisType.Anomaly }

};

var reportData = new ReportData

{

WhitelabelId = "test-whitelabel"

};

var insights = new List<Insight>

{

new Insight { Id = "insight1", Title = "Test Insight" }

};

\_mockReportService

.Setup(s => s.GetReportData(request.ReportId, request.Parameters))

.ReturnsAsync(reportData);

\_mockAnalyticsEngine

.Setup(e => e.AnalyzeData(reportData, request.AnalysisTypes))

.ReturnsAsync(insights);

// Act

await \_insightService.GenerateInsights(request);

// Assert

\_mockRepository.Verify(r => r.StoreInsights(insights), Times.Once);

}

}

## **6. Deployment Strategy**

1. **Containerization**:  
   * Package the insight service components in separate Docker containers
   * Use Kubernetes for orchestration and scaling
   * Implement resource limits and requests for predictable performance
2. **Scaling Approach**:  
   * Horizontally scale the analysis engine for processing capacity
   * Use Redis for distributed locking to prevent duplicate processing
   * Implement separate scaling policies for background processing and API services
3. **Monitoring & Observability**:  
   * Add detailed logging for insight generation process
   * Create custom metrics for tracking insight quality and user engagement
   * Set up alerts for insight generation pipeline failures
4. **Rollout Strategy**:  
   * Deploy to staging environment for initial testing
   * Enable feature for a subset of users with feature flags
   * Monitor performance metrics and user feedback
   * Roll out to all users after validation

## **Implementation Roadmap**

| **Phase** | **Timeline** | **Key Deliverables** |
| --- | --- | --- |
| 1. Foundation | Weeks 1-2 | - Database schema & stored procedures<br>- Core service interfaces<br>- Basic insight repository implementation |
| 2. Analysis Engine | Weeks 3-4 | - Trend & anomaly analyzers<br>- Analytics engine implementation<br>- Test suite for analysis components |
| 3. Frontend Components | Weeks 5-6 | - Insights panel & cards<br>- Redux integration<br>- Insight detail view with visualization |
| 4. Real-time Updates | Weeks 7-8 | - SignalR notification hub<br>- Background service for scheduled analysis<br>- ML service integration |
| 5. Performance & Testing | Weeks 9-10 | - Performance optimization<br>- End-to-end testing<br>- Monitoring & observability |

This comprehensive implementation provides a fully functional AI insights pipeline that will enhance the ProgressPlay Reporting Platform with automated, intelligent data analysis and meaningful user-facing insights.