¡Excelente! Ya vi tu proyecto React y el paquete del modelo **jurisdiction\_model** (es un modelo de **ML.NET** con una TransformerChain). Te dejo un plan claro y el código que necesitas para que en **cada pantalla** se diferencien y filtren las alertas **TCS** vs **NO TCS**, usando ese modelo.

**1) Flujo recomendado (end-to-end)**

1. **Backend .NET (API)**
   * Carga **una sola vez** el modelo jurisdiction\_model al iniciar la app.
   * Expone dos endpoints:
     + GET /OpenProblemIds?environment=... (como ya tienes).
     + GET /{id} que ahora **devuelve también** jurisdiction con:  
       { isTcs: boolean, label: "TCS"|"NO TCS", confidence: 0..1, features: { ... } }.
   * (Opcional) Endpoint directo: POST /jurisdiction/predict para probar textos sueltos.
2. **Frontend React**
   * Tus páginas ya están separadas: **/tcs**, **/otros**, **/mock**.
   * Con lo que devuelva el backend (campo problem.jurisdiction.isTcs), las páginas:
     + **TCSProblems.jsx**: muestran isTcs === true (o isTcs !== false, como ya lo haces).
     + **OtherProblems.jsx**: muestran isTcs === false.
   * Añadir:
     + Badges “TCS” / “NO TCS” + % confianza en cada card.
     + Contadores en la barra de navegación.
     + Un toggle “Todos / TCS / NO TCS” en una página “All” si lo deseas.

Con esto, no tendrás que “adivinar” en el cliente: el **backend** etiqueta y el **frontend** solo muestra/filtra.

**2) Backend .NET — integrar el modelo ML.NET**

Asumo ASP.NET Core minimal API o controlador. Si usas otra estructura, te dejo la idea y el código es muy fácil de adaptar.

**2.1. Clases de entrada/salida del modelo**

// Models/JurisdictionInput.cs

public class JurisdictionInput

{

// Ajusta los nombres a lo que realmente alimentaste al entrenar:

public string Title { get; set; }

public string Tenant { get; set; }

public string Environment { get; set; }

public string AffectedCiJoined { get; set; } // "CI1;CI2;CI3"

}

// Models/JurisdictionOutput.cs

public class JurisdictionOutput

{

public bool IsTcs { get; set; }

public string Label { get; set; } // "TCS" o "NO TCS"

public float Confidence { get; set; } // 0..1

public Dictionary<string, object> Features { get; set; } = new();

}

// DTO que devuelve tu endpoint GET /{id}

public class ProblemDto

{

// ... todos tus campos actuales:

public string ProblemId { get; set; }

public string Title { get; set; }

public string Tenant { get; set; }

public string Environment { get; set; }

public IEnumerable<string> AffectedCI { get; set; } = Enumerable.Empty<string>();

// Nuevo:

public JurisdictionOutput Jurisdiction { get; set; }

}

**2.2. Cargar el modelo como Singleton**

// Services/JurisdictionPredictor.cs

using Microsoft.ML;

public interface IJurisdictionPredictor

{

JurisdictionOutput Predict(JurisdictionInput input);

}

public class JurisdictionPredictor : IJurisdictionPredictor

{

private readonly MLContext \_ml;

private readonly ITransformer \_model;

private readonly PredictionEngine<JurisdictionInput, ModelPrediction> \_engine;

// Clase técnica usada para mapear la salida del modelo entrenado

private class ModelPrediction

{

public float[] Score { get; set; } // Probabilidades por clase

public uint PredictedLabel { get; set; } // depende del modelo; si usaste Key

public string PredictedLabelValue { get; set; } // si tu pipeline preserva string

}

public JurisdictionPredictor(IConfiguration config)

{

\_ml = new MLContext(seed: 1);

var modelPath = config["JurisdictionModel:Path"]; // e.g. "jurisdiction\_model"

if (string.IsNullOrWhiteSpace(modelPath))

throw new InvalidOperationException("Falta JurisdictionModel:Path en appsettings.");

// ML.NET puede cargar un directorio de modelo (carpeta con TransformerChain)

using var fs = File.OpenRead(Path.Combine(modelPath, "TransformerChain", "Model.key"));

// Nota: si tu modelo fue guardado como un .zip, usa esa ruta en su lugar.

\_model = \_ml.Model.Load(fs, out \_);

\_engine = \_ml.Model.CreatePredictionEngine<JurisdictionInput, ModelPrediction>(\_model);

}

public JurisdictionOutput Predict(JurisdictionInput input)

{

var pred = \_engine.Predict(input);

// Mapea las clases. Ajusta estas líneas según tu entrenamiento:

// Supongamos dos clases: index 0 = "NO TCS", index 1 = "TCS"

float probNoTcs = pred.Score.Length > 0 ? pred.Score[0] : 0;

float probTcs = pred.Score.Length > 1 ? pred.Score[1] : 0;

bool isTcs = probTcs >= probNoTcs;

float confidence = Math.Max(probTcs, probNoTcs);

return new JurisdictionOutput

{

IsTcs = isTcs,

Label = isTcs ? "TCS" : "NO TCS",

Confidence = confidence,

Features = new Dictionary<string, object>

{

["probTcs"] = probTcs,

["probNoTcs"] = probNoTcs

}

};

}

}

🔧 **Importante**: Si tu modelo fue guardado como **MLModel.zip**, cambia el Load a esa ruta.  
Si está como carpeta (como el que subiste), necesitas apuntar al archivo correcto. A veces ML.NET espera un **.zip**; si lo tienes en carpeta, considera re-exportarlo como zip o cargar con el runner que usaste al entrenar.

**2.3. Registrar servicio y configuración**

**appsettings.json**

{

"JurisdictionModel": {

"Path": "jurisdiction\_model" // pon la ruta donde despliegas esa carpeta/zip

}

}

**Program.cs**

builder.Services.AddSingleton<IJurisdictionPredictor, JurisdictionPredictor>();

**2.4. Usar el predictor al construir el ProblemDto**

En tu controlador/handler actual de GET /{id}:

[HttpGet("{id}")]

public async Task<ActionResult<ProblemDto>> GetById(string id, [FromServices] IJurisdictionPredictor predictor)

{

var raw = await \_dynatraceService.GetProblemAsync(id); // tu fetch actual

var input = new JurisdictionInput

{

Title = raw.Title ?? "",

Tenant = raw.Tenant ?? "",

Environment = raw.Environment ?? "",

AffectedCiJoined = string.Join(";", raw.AffectedCI ?? Array.Empty<string>())

};

var jurisdiction = predictor.Predict(input);

var dto = new ProblemDto

{

ProblemId = raw.ProblemId,

Title = raw.Title,

Tenant = raw.Tenant,

Environment = raw.Environment,

AffectedCI = raw.AffectedCI,

Jurisdiction = jurisdiction

};

return Ok(dto);

}

(Optativo) endpoint de prueba:

[HttpPost("jurisdiction/predict")]

public ActionResult<JurisdictionOutput> Predict([FromBody] JurisdictionInput input,

[FromServices] IJurisdictionPredictor predictor)

{

return Ok(predictor.Predict(input));

}

**3) Frontend React — mostrar y filtrar TCS vs NO TCS**

Tu app **ya** filtra con p.jurisdiction?.isTcs. Hagamos tres mejoras rápidas:

**3.1. Badge en la card**

src/components/ProblemCard.jsx (añade el badge arriba del título)

// dentro del componente:

const isTcs = problem?.jurisdiction?.isTcs;

const confidence = problem?.jurisdiction?.confidence;

const badgeStyle = {

padding: '0.25rem 0.5rem',

borderRadius: '999px',

fontWeight: 700,

fontSize: '0.8rem',

background: isTcs ? '#1b5e20' : '#b71c1c',

color: '#fff',

marginRight: '0.5rem'

};

...

<div style={{ display: 'flex', alignItems: 'center', marginBottom: '8px' }}>

<span style={badgeStyle}>{isTcs ? 'TCS' : 'NO TCS'}</span>

{typeof confidence === 'number' && (

<span style={{ opacity: 0.75 }}>

Confianza: {(confidence \* 100).toFixed(1)}%

</span>

)}

</div>

<h3 style={{ margin: 0 }}>{title}</h3>

**3.2. Contadores en la Navbar**

src/components/Navbar.jsx — agrega contadores usando un *context* o prop.  
Como ya tienes useOpenProblems, podemos colocar los contadores en cada página, o crear un **CountsContext**. La opción simple: **mostrarlo en cada página** arriba de la lista:

// En TCSProblems.jsx (y similar en OtherProblems.jsx)

const total = problems.length;

const tcsCount = problems.filter(p => p.jurisdiction?.isTcs === true).length;

const noTcsCount = problems.filter(p => p.jurisdiction?.isTcs === false).length;

<div style={{ display:'flex', gap:'12px', margin:'0 0 12px 0' }}>

<div><strong>Total:</strong> {total}</div>

<div><strong>TCS:</strong> {tcsCount}</div>

<div><strong>NO TCS:</strong> {noTcsCount}</div>

</div>

Si quieres los **píldoras en la Navbar**, mueve estos conteos a un contexto global (por ejemplo ProblemsCountsContext) y consúmelos en Navbar.

**3.3. Nueva página “All” con toggle**

Crea src/pages/AllProblems.jsx:

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

import useOpenProblems from '../hooks/useOpenProblems';

import ProblemCard from '../components/ProblemCard';

import UsernameInput from '../components/UsernameInput';

import { useUser } from '../context/UserContext';

export default function AllProblems() {

const { problems, loading, error } = useOpenProblems(5000);

const { username } = useUser();

const [filter, setFilter] = useState('ALL'); // 'ALL' | 'TCS' | 'NO\_TCS'

const filtered = useMemo(() => {

if (filter === 'TCS') return problems.filter(p => p.jurisdiction?.isTcs === true);

if (filter === 'NO\_TCS') return problems.filter(p => p.jurisdiction?.isTcs === false);

return problems;

}, [problems, filter]);

return (

<div style={{ maxWidth: 900, margin:'0 auto', padding:'1rem' }}>

<h1>Todos los problemas</h1>

<UsernameInput/>

<div style={{ display: 'flex', gap: 8, margin:'12px 0' }}>

<button onClick={() => setFilter('ALL')}>Todos</button>

<button onClick={() => setFilter('TCS')}>Solo TCS</button>

<button onClick={() => setFilter('NO\_TCS')}>Solo NO TCS</button>

</div>

{filtered.map(item => (

<ProblemCard key={item.problemId} problem={item} username={username} />

))}

</div>

);

}

Y en App.jsx agrega la ruta:

<Route path="/all" element={<AllProblems />} />

Y en Navbar.jsx agrega el link:

<NavLink to="/all" className={...}>Todos</NavLink>

**4) ¿Y si el backend aún no devuelve jurisdiction?**

Mientras conectas el modelo en el backend, puedes hacer un **fallback temporal** en el cliente para no frenar:

* Si existe problem.jurisdiction, úsalo.
* Si no existe, aplica una **heurística simple** (temporal) en el cliente, por ejemplo:
  + Si tenant o affectedCI contiene claves de tus clientes TCS → isTcs = true.
  + Si title contiene “external”, “third-party”, etc. → isTcs = false.

Ejemplo en useOpenProblems.js (después de fetchProblemById):

function heuristicJurisdiction(p) {

if (p.jurisdiction) return p.jurisdiction;

const text = `${p.title} ${p.tenant} ${(p.affectedCI || []).join(' ')}`.toLowerCase();

const looksTcs = /tcs|ecbp|poe|clon\_poe|vios|vmware/.test(text); // ajusta a tus claves reales

return {

isTcs: looksTcs,

label: looksTcs ? 'TCS' : 'NO TCS',

confidence: 0.55

};

}

// cuando creas el objeto final:

const full = await fetchProblemById(id);

full.jurisdiction = heuristicJurisdiction(full);

(Esto es **solo mientras** conectas el modelo real.)

**5) Validación rápida que ya tienes en el Front**

Revisé tus páginas:

* TCSProblems.jsx ya filtra con p.jurisdiction?.isTcs !== false.
* OtherProblems.jsx debe filtrar con p.jurisdiction?.isTcs === false.
* ProblemCard.jsx muestra bien el contenido; solo añade el **badge** y % de confianza.
* utils/config.js obtiene API\_BASE\_URL. Asegúrate que el **GET /{id}** ya venga con el bloque jurisdiction.

**Ejemplo de JSON esperado para GET /{id}:**

{

"problemId": "12345",

"title": "CPU high on server ABC",

"tenant": "CocaCola-EC",

"environment": "Productivo",

"affectedCI": ["SRV-ABC", "SRV-XYZ"],

"jurisdiction": {

"isTcs": true,

"label": "TCS",

"confidence": 0.91,

"features": { "probTcs": 0.91, "probNoTcs": 0.09 }

}

}

**6) Checklist para pasar a producción**

* **Ruta del modelo** correcta en appsettings.json (zip vs carpeta).
* Cargar el modelo **una vez** (Singleton) y **reusar** el PredictionEngine (o PredictionEnginePool si vas a alta concurrencia).
* Endpoint GET /{id} devuelve jurisdiction.
* React: badge + porcentajes, contadores, rutas funcionando.
* (Opcional) Página “Todos” con toggle.
* Logs: registra problemId, label, confidence (útil para auditar).
* Métrica: agrega en tu tablero un widget con **% TCS vs NO TCS**, y distribución de confianza.

Si quieres, puedo adaptar exactamente el **controlador concreto** de tu API (pegas aquí tu Controller/Minimal API) y te devuelvo el archivo completo con la carga del **jurisdiction\_model** integrada y probada a nivel de compilación. También puedo añadir el **badge** y los **contadores** en tu React y pasarte los archivos listos para reemplazar.

##### Dijiste:

SI AYUDAME COMPLETO Y PASO A PASO LO QUE HACES

###### ChatGPT dijo:

Pensado durante 54s

¡Listo, Dennys! Te dejo un plan **completo y paso a paso**, con **archivos descargables** para que pegues y pruebes. Vamos a lograr que **cada pantalla** muestre claramente qué alertas son **TCS** y cuáles **NO TCS**, usando tu modelo **jurisdiction\_model**.

# 🧭 Resumen del enfoque

1. **Backend (.NET/ML.NET)**  
   Cargamos tu modelo una sola vez y, al responder GET /{id}, agregamos un bloque:
2. "jurisdiction": { "isTcs": true|false, "label": "TCS|NO TCS", "confidence": 0..1 }

Así el **frontend** solo filtra y pinta.

1. **Frontend (React)**
   * Agregamos un **badge** “TCS/NO TCS” + % de **confianza** en cada tarjeta.
   * Añadimos una página “**Todos**” con **toggle** (Todos / TCS / NO TCS).
   * (Opcional) Contadores por tipo.

# ✅ Lo que te entrego listo para bajar

* **Backend skeleton (NET 8 + ML.NET)** con endpoints compatibles y predictor integrado:  
  **Descargar JurisdictionApiSkeleton.zip**

Úsalo tal cual o copia solo el servicio del predictor a tu API existente.

* **Helpers para React** (página All + patch de la card + README):  
  **Descargar React\_TCS\_NOTCS\_helpers.zip**

# 🛠️ Paso a paso

## Paso 1 — Integrar el modelo en el backend

### Opción A (recomendada): ****Añadir predictor a tu API actual****

1. **Crea** una clase de servicio JurisdictionPredictor y **regístrala** como Singleton.
2. **Carga** tu modelo ML.NET (ideal: exportado como **.zip**).
3. **Modifica** tu handler de GET /{id} para agregar jurisdiction al JSON que ya envías.

Te dejo el código listo (viene en el ZIP), copiando lo esencial:

**Program.cs**

builder.Services.AddSingleton<IJurisdictionPredictor, JurisdictionPredictor>();

**Services/JurisdictionPredictor.cs** (usa ML.NET)

public interface IJurisdictionPredictor

{

JurisdictionOutput Predict(JurisdictionInput input);

}

public class JurisdictionPredictor : IJurisdictionPredictor

{

private readonly PredictionEngine<JurisdictionInput, ModelPrediction> \_engine;

public JurisdictionPredictor(IConfiguration config, ILogger<JurisdictionPredictor> logger)

{

var zipPath = config["JurisdictionModel:ZipPath"]; // ej: ./jurisdiction\_model.zip

using var fs = File.OpenRead(zipPath);

var ml = new MLContext();

var model = ml.Model.Load(fs, out \_);

\_engine = ml.Model.CreatePredictionEngine<JurisdictionInput, ModelPrediction>(model);

}

public JurisdictionOutput Predict(JurisdictionInput input)

{

var pred = \_engine.Predict(input);

// Suponemos Score[0]=NO TCS, Score[1]=TCS (ajústalo si tu modelo difiere)

var noTcs = pred.Score?.Length > 0 ? pred.Score[0] : 0f;

var tcs = pred.Score?.Length > 1 ? pred.Score[1] : 0f;

return new JurisdictionOutput {

IsTcs = tcs >= noTcs,

Label = (tcs >= noTcs) ? "TCS" : "NO TCS",

Confidence = Math.Max(tcs, noTcs),

Features = new() { ["probTcs"] = tcs, ["probNoTcs"] = noTcs }

};

}

private sealed class ModelPrediction

{

public float[]? Score { get; set; }

public string? PredictedLabel { get; set; }

}

}

**appsettings.json**

{

"JurisdictionModel": { "ZipPath": "./jurisdiction\_model.zip" }

}

**En tu GET /{id}**, agrega:

var input = new JurisdictionInput {

Title = raw.Title ?? "",

Tenant = raw.Tenant ?? "",

Environment = raw.Environment ?? "",

AffectedCiJoined = string.Join(";", raw.AffectedCI ?? Array.Empty<string>())

};

var j = predictor.Predict(input);

raw.Jurisdiction = j; // ← añade el bloque al JSON que devuelves

return Ok(raw);

💡 **Si tu modelo NO está en .zip**, te recomiendo **re-exportarlo** a un solo archivo .zip (lo más estable con ML.NET). Si necesitas, te guío para empaquetarlo.

### Opción B: ****Usar el skeleton**** que te dejé

1. Descarga y descomprime: **JurisdictionApiSkeleton.zip**.
2. Copia tu jurisdiction\_model.zip junto al Program.cs o ajusta JurisdictionModel:ZipPath.
3. Edita appsettings.json → DynatraceApi:BaseUrl para apuntar a tu backend/DT actual que ya responde /OpenProblemIds y /{id}.
4. Corre:

dotnet restore

dotnet run --project src/JurisdictionApi/JurisdictionApi.csproj

1. Abre Swagger y prueba /jurisdiction/predict y /{id}.

Este skeleton hace **passthrough** de tus endpoints existentes y **inyecta** jurisdiction al GET /{id}.

## Paso 2 — Cambios en el frontend React

1. **Añade** la nueva página **Todos** (con filtros).
   * Copia AllProblems.jsx a:  
     react-dynatrace-app/src/pages/AllProblems.jsx  
     (el archivo viene en **React\_TCS\_NOTCS\_helpers.zip**)
2. **Patch a la tarjeta** para mostrar **badge TCS/NO TCS** y **confianza**.
   * Aplica ProblemCard.jsx.patch o edítalo manualmente:
     + Calcula isTcs y confidence desde problem.jurisdiction.
     + Pinta un **pill** verde/rojo y el porcentaje.
3. **Rutas y navbar**:
   * En src/App.jsx:
   * import AllProblems from './pages/AllProblems';
   * // ...
   * <Route path="/all" element={<AllProblems />} />
   * En src/components/Navbar.jsx:
   * <NavLink to="/all" className={({ isActive }) => isActive ? 'nav-item active' : 'nav-item'}>Todos</NavLink>
4. **(Opcional) Contadores**
   * Ya te los dejé en la página AllProblems.jsx (Total/TCS/NO TCS).
   * Puedes replicarlos en TCSProblems.jsx y OtherProblems.jsx.
5. **Config**
   * Verifica que src/utils/config.js apunte a tu **API\_BASE\_URL** del backend que ahora devuelve jurisdiction dentro de cada problema.

Todo esto está documentado en **React\_TCS\_NOTCS\_README.md** dentro del zip de helpers.

## Paso 3 — Pruebas de punta a punta

1. Levanta el **backend** (tu API o el skeleton).
2. En **React**, ejecuta: npm install → npm run dev (o tu script).
3. Entra a **/all**, **/tcs**, **/otros** y verifica:
   * Cada tarjeta tiene badge **TCS**/**NO TCS** y **Confianza %**.
   * Las páginas filtran correctamente según jurisdiction.isTcs.
   * Los contadores reflejan los totales.

## Paso 4 — Fallback temporal (si el backend aún no devuelve jurisdiction)

Mientras integras el predictor, puedes aplicar una **heurística suave** en el cliente (solo de forma temporal):

// en useOpenProblems.js, después de fetchProblemById:

function heuristicJurisdiction(p) {

if (p.jurisdiction) return p.jurisdiction;

const text = `${p.title} ${p.tenant} ${(p.affectedCI || []).join(' ')}`.toLowerCase();

const looksTcs = /tcs|ecbp|poe|clon\_poe|vios|vmware/.test(text); // ajusta a tus claves reales

return { isTcs: looksTcs, label: looksTcs ? 'TCS' : 'NO TCS', confidence: 0.55 };

}

full.jurisdiction = heuristicJurisdiction(full);

# 🧪 Buenas prácticas y validación

* **CORS** habilitado en backend para que el React pueda llamar sin bloqueo.
* **Logs** del predictor: guarda problemId, label, confidence para auditoría.
* **Métricas**: agrega un widget en tu dashboard con % de TCS/NO TCS y distribución de confianza (0.5–0.7, 0.7–0.9, >0.9).
* **Rendimiento**: el predictor está en **Singleton**; si necesitas alta concurrencia, migra a PredictionEnginePool.

## ¿Qué sigue?

Si me compartes **tu controlador/handler real** de GET /{id} (o Program.cs), te lo **devuelvo completo** con las líneas **exactas** insertadas (sin mover nada más).  
¿Quieres que deje también lista una **métrica en tu UI** (ej. chip con conteo TCS/NO TCS en la Navbar)? Te la agrego enseguida.