PORTFOLIO OF EVIDENCE

RUAN LEENDERS 578427

3 September 2025-3 March 2026

Work & Experience Portfolio

Contents

1. EXPERIENCES AND ACHIEVEMENTS 7

MAN TCO 7

Configurations: 7

TASK 1: R&M Matrix editable, loading all models from Vehicle Management with Excel R&M Rates (ratios are editable). 7

TASK 2: RV Table editable, loading all models from Vehicle Management with Excel RV values. 9

Deal Owner: 11

TASK 3: Colour coding in User Management to identify owners and their deals. 11

TASK 4: Deal owner column added to Deal Reports table. 14

TASK 5: Deal owner specified in the TCO Workflow when editing. 16

TASK 6: Dropdown display to assign a different owner to an existing deal. 17

Optimization: 19

TASK 7: Compressed background image to load faster. 19

TASK 8: Optimized queries for load optimization. 21

Services: 23

TASK 9: Password reset fixed, and signup forum removed (Add user in user management their default password is TCO-M*N-2025!sb, they can then reset it). 23

TASK 10: PDF generation of mapping fixed (Generate Quote). 25

MAN SPR 28

2.1.1 About 28

2.1.2 Market Opportunity 28

2.1.3 Technical Architecture 28

2.1.4 Technology Stack 29

2.1.5 Conclusion 29

2.1.6 Summary 30

PROJECT 1: SPR Multi-Approver Workflow Implementation Documentation 35

Project Overview 35

TASK 1: Multi-Approver Database Schema and Model Updates 35

Why This Was Needed: 36

TASK 2: Custom Price Range Configuration for SPR Approvers 36

Function Explanation: 37

Why This Was Needed: 37

TASK 3: Multi-Approver Assignment at Vetter Credit Review Stage 38

Function Explanation: 38

Why This Was Needed: 38

TASK 4: Multi-Approver Approval Workflow Logic 38

Function Explanation: 39
TASK 5: Multi-Approver Rejection Workflow 40
Function Explanation: 40
Why This Was Needed: 40
TASK 6: Dynamic Approve Button Text Based on Approval Status 41
Function Explanation: 41
Why This Was Needed: 41
TASK 7: Multi-Approver Status Display in UI 42
Function Explanation: 42
Why This Was Needed: 42
TASK 8: Todo List and Watch List Updates for Multi-Approvers 43
Function Explanation: 43
Function Explanation: 44
Why This Was Needed: 44
Summary 45
PROJECT 2: MAN Trucks & Bus TCO Tool 45
TASK 1: Fuel and AdBlue Calculation Reactivity Fix for MAN 1 45
Function Explanation: 46
Why This Was Needed: 46
TASK 2: Total Fuel Consumption Display Correction 47
Function Explanation: 47
Why This Was Needed: 47
TASK 3: Competitor Fuel + AdBlue CPK Calculation Fix 48
Function Explanation: 48
Why This Was Needed: 48
TASK 4: Competitor Total Cost Per Kilometer Enhancement 49
Function Explanation: 49
Why This Was Needed: 49
TASK 5: Competitive Analysis Section Overhaul 50
Function Explanation: 50
Why This Was Needed: 50
PROJECT 1: MAN TCO 51
TASK 1: Extras tick box to control quote display (non-ticked extras will not appear on quote) ... 51
TASK 2: Total GP color coding (red when negative) 53
TASK 3: Duplicate MAN Vehicle (MAN 2+) inherits MAN 1 values, independently configurable . 54
TASK 4: Competitive Analysis displays MAN 2 comparisons alongside MAN 1 and Competitors 56
TASK 5: Enhanced MFS details with deposit and interest rate display 59
TASK 6: Universal interest rate configuration (configurable, non-editable in workflow) 60

TASK 7: Salesman can decrease trade back value without creating negative provisions 63
TASK 8: Increased trade back value creates appropriate provision 65
TASK 9: Decreased trade back value does not affect provisions (no negative provisions) 66
TASK 10: Stand-in value equals offer to customer (no credit given on loading) 67
TASK 11: Added UI user messages and guidance to Trade Back workflow step 68
PROJECT 2: MAN CONTRACT MANAGEMENT 70
Project Overview and Business Context 70
Technology Stack and Architecture 71
Database Schema and Migrations 71
Task 1: Contract Creation Workflow 71
Task 2: Workflow Stage Management 71
Task 3: Task Management System 71
Task 4: Digital Signature Implementation 72
Task 5: Internal Signature Workflow 72
Task 6: External Signature Workflow 72
Task 7: Contract Management and Viewing 72
Task 8: User Authentication and Authorization 72
Task 9: Admin Panel Features 72
Task 10: Document Management 73
Task 11: Reporting and Export 73
Task 12: Repository and Templates 73
Task 13: Bug Fixes and Critical Issues Resolved 73
Task 14: Database Operations and Testing 73
Task 15: UI and Styling Improvements 73
Development Process and Methodologies 73
Key Achievements and Deliverables 74
Technical Skills Demonstrated 74
NEIGHBORHOOD SECURITY APP 76
What we do 76
The problem we solve 76
How it works 76
Technology and build 76
Languages and codebase 76
Why it matters 76
Feature 1: Snapshot image on alert detailDescription 77
Where it appears 77
Database 77
How it works (flow) 77

Code 77
Tested / status 77
Feature 2: Vehicle profile (score trend + prior alerts) 79
Description 79
Where it appears 79
Database 79
How it works (flow) 79
Code 79
Tested / status 80
Feature 3: Neighbourhood Admin – Manage Members (search, filters, household dropdown) 81
Description 81
Where it appears 81
Database 81
How it works (flow) 81
Code 82
Tested / status 82
Feature 4: Vote reason dropdown 86
Description 86
Where it appears 86
Database 86
How it works (flow) 86
Code 86
Tested / status 86
Feature 5: Realtime vote counts 88
Description 88
Where it appears 88
Database 88
How it works (flow) 88
Code 88
Tested / status 88
Feature 6: Doc approval (admin) 89
Description 89
Database 89
How it works (flow) 89
Code 89
Feature 7: Admin face approval – test steps 91
Description 91
Where to test 91

Test steps 91
Database 91
Tested / status 91
Feature 8: Face quality checks – test steps 92
Description 92
Where to test 92
Test steps 92
Code 92
Tested / status 92
Feature 9: Push notifications – test steps 92
Description 92
Where to test 92
Requirements 92
Test steps 92
If push notifications don't appear 93
Code 93
Tested / status 93
2. WORKPLACE JOURNEY AND REFLECTIONS 94
2.1 Significant experiences and challenges 94
2.2 Lessons learned and skills gained 94
Technical 94
Process 94
Growth 94
3. MENTOR FEEDBACK, WORK ETHICS, AND SOFT SKILLS 95
4. CONCLUSION 98

1. EXPERIENCES AND ACHIEVEMENTS

MAN TCO

Configurations:

TASK 1: R&M Matrix editable, loading all models from Vehicle Management with Excel R&M Rates (ratios are editable).

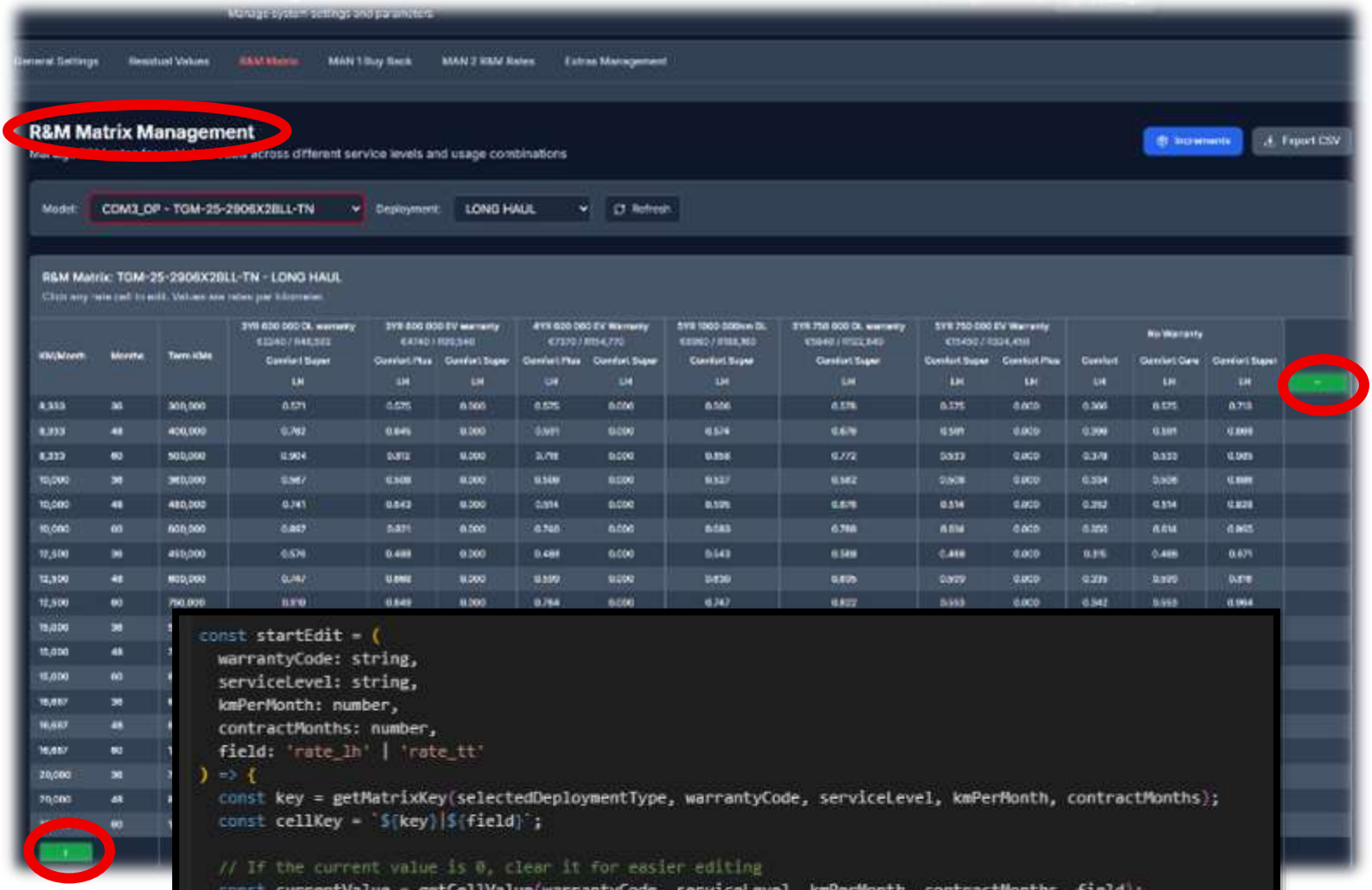
Completed the development of editable R&M Matrix and Residual Value Table systems with Excel-compatible data import/export functionality, enabling administrators to manage vehicle maintenance rates and residual values efficiently. The R&M Matrix implementation required resolving critical Row Level Security (RLS) issues that were preventing data access. The system was transformed from a read-only configuration table to a fully editable management interface where administrators can modify maintenance rate ratios directly within the application.

The R&M Matrix system now dynamically loads all vehicle models from the Vehicle Management table, ensuring that the matrix always reflects the current vehicle model database. Previously, the system relied on hardcoded model lists that required code updates whenever new vehicle models were added. This was fixed by implementing dynamic model loading that queries the vehicle_models table at runtime, automatically including all active vehicle models in the R&M Matrix management interface.

The Excel-compatible R&M Rates functionality was implemented to allow administrators to edit maintenance rate ratios (rate_lh and rate_tt) for each vehicle model, deployment type, warranty option, service level, and usage combination. The interface displays rates in an Excel-like spreadsheet format with warranty types across the top columns and usage parameters (kilometers per month, contract months) as row headers. Each rate cell is individually editable, allowing administrators to adjust maintenance costs without requiring database access or code modifications.

Critical fixes included resolving RLS policy issues on the rm_matrix_tables table that were causing 406 (Forbidden) errors when attempting to read or update matrix data. The policies were restructured to allow authenticated users to read, insert, update, and delete R&M matrix entries while maintaining appropriate security boundaries. Additionally, RLS policies on the vehicle_models table were updated to ensure the R&M Matrix management interface could access vehicle model data for dynamic loading.

The editable ratio system enables real-time updates to maintenance rates, with changes immediately reflected throughout the TCO calculation system. Administrators can now adjust maintenance costs for specific vehicle configurations, deployment scenarios, warranty types, and service levels directly through the user interface, eliminating the need for manual database updates or system redeployment when maintenance pricing changes occur.




```
const startEdit = (
  warrantyCode: string,
  serviceLevel: string,
  kmPerMonth: number,
  contractMonths: number,
  field: 'rate_lh' | 'rate_tt'
) => {
  const key = getMatrixKey(selectedDeploymentType, warrantyCode, serviceLevel, kmPerMonth, contractMonths);
  const cellKey = `${key}|${field}`;

  // If the current value is 0, clear it for easier editing
  const currentValue = getCellValue(warrantyCode, serviceLevel, kmPerMonth, contractMonths, field);
  if (typeof currentValue === 'number' && currentValue === 0) {
    setPendingChanges(prev => {
      const newChanges = new Map(prev);
      const existing = newChanges.get(cellKey) || {};
      newChanges.set(cellKey, { ...existing, [field]: '' as any });
      return newChanges;
    });
  }

  setEditingCells(prev => new Set([...prev, cellKey]));
};

const handleCellChange = (
  warrantyCode: string,
  serviceLevel: string,
  kmPerMonth: number,
  contractMonths: number,
  field: 'rate_lh' | 'rate_tt',
  value: string
) => {
  const key = getMatrixKey(selectedDeploymentType, warrantyCode, serviceLevel, kmPerMonth, contractMonths);
  const cellKey = `${key}|${field}`;
  const numValue = parseFloat(value) || 0;

  setPendingChanges(prev => {
    const newChanges = new Map(prev);
    const existing = newChanges.get(cellKey) || {};
    newChanges.set(cellKey, { ...existing, [field]: numValue });
    return newChanges;
  });
```

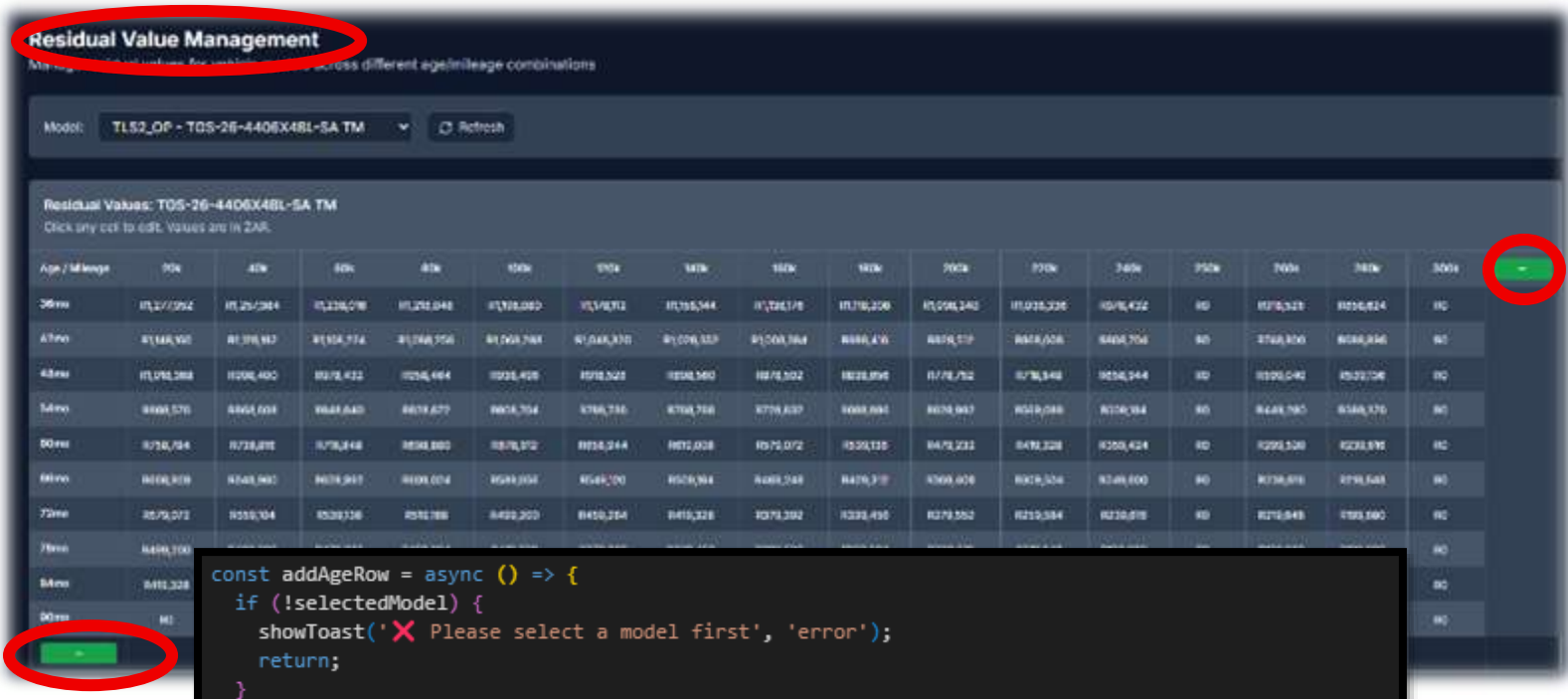
The startEdit function begins the edit session for a specific cell. It builds a unique key combining the deployment type, warranty code, service level, kilometers per month, contract months, and field type (rate_lh or rate_tt). This key identifies the exact cell being edited. If the current value is zero, it clears the cell (sets it to an empty string) to make input easier. It updates pendingChanges immutably by copying the Map, preserving existing changes, and then updates the cell's field. Finally, it adds the cell key to editingCells to mark it as active. When the user types, handleCellChange captures the input. It uses the same key generation logic to identify the cell. It parses the string input into a number using parseFloat, defaulting to zero if parsing fails. It updates pendingChanges by creating a new Map, preserving any existing changes for that cell, and merging the new value using object spread. This keeps changes in memory without writing to the database. Together, they provide an Excel-like editing flow: clicking a cell calls startEdit to enter edit mode and track the cell, typing calls handleCellChange to update pending values, and saving commits all pending changes to the database. The cell key format ensures each editable cell has a unique identifier across the matrix. Immutable state updates avoid React rendering issues, and the pending changes pattern lets users make multiple edits before saving, improving the editing experience.

TASK 2: RV Table editable, loading all models from Vehicle Management with Excel RV values.

The Residual Value (RV) Table system received similar enhancements, transforming it from a static configuration table into a fully editable management interface with Excel-compatible residual value data. The implementation required resolving data mapping issues between the RV table's full model codes (such as TGS-26-4406X4BL-SA) and the Vehicle Management table's short codes (such as TLS2_OP). This was fixed by implementing a bidirectional mapping system that automatically converts between the two code formats, ensuring seamless integration between the RV table and the Vehicle Management database. The RV Table now dynamically loads all vehicle models from the Vehicle Management table at runtime, automatically including all active vehicle models regardless of whether they currently have residual value data configured. Previously, the system only displayed models that already had RV data entries, making it impossible to add residual values for new vehicle models without manual database intervention.

The Excel-compatible RV values functionality was implemented to allow administrators to edit residual value percentages for each vehicle model across different age and mileage combinations. The interface displays residual values in a spreadsheet-like matrix format with age in months as row headers and mileage in kilometers as column headers. Each cell in the matrix represents the residual value percentage for a specific combination of vehicle age and total mileage, allowing administrators to adjust depreciation calculations directly within the application without requiring database access or code modifications. The system supports dynamic addition of new age and mileage parameters, enabling administrators to extend the matrix as business requirements evolve, while maintaining the Excel-like editing experience where users can click on any cell to modify its value.

Critical improvements included implementing intelligent model code mapping that handles the conversion between RV table format codes and Vehicle Management format codes automatically. When a model exists in both systems with different code formats, the system maps them correctly, ensuring that residual value calculations reference the correct vehicle model from the Vehicle Management database. The dynamic loading mechanism merges models that already have RV data with models that don't, presenting a comprehensive list that includes all available vehicle models while preserving existing RV data associations. This allows administrators to view and edit residual values for existing models while also identifying models that need RV data to be created.




```
const addAgeRow = async () => {
  if (!selectedModel) {
    showToast('❌ Please select a model first', 'error');
    return;
  }

  try {
    setLoading(true);
    // Use RV code (full code) for database operations
    const rvCode = VEHICLE_TO_RV_CODE_MAP[selectedModel] || selectedModel;
    const models = [rvCode];
    const mileageKms = rvMatrix.mileageKms.length > 0 ? rvMatrix.mileageKms : DEFAULT_MILEAGES;
    await residualValueService.addAgeRow(newAge, models, mileageKms);
    showToast(`✅ Added ${newAge} months row`, 'success');
    setShowAddAge(false);
    setNewAge(90);
    loadRVMatrix();
  } catch (error) {
    console.error('Error adding age row:', error);
    showToast('❌ Error adding age row', 'error');
  } finally {
    setLoading(false);
  }
};
```

The addAgeRow function adds a new age row to the RV Table matrix. It first checks that a model is selected; if not, it shows an error and exits. If a model is selected, it sets a loading state, then converts the selected Vehicle Management short code to the RV table full code via VEHICLE_TO_RV_CODE_MAP, falling back to the selected model code if no mapping exists. It determines the mileage values to use: existing mileage columns if present, otherwise default mileage values. It calls residualValueService.addAgeRow with the new age value, the converted model code, and the mileage array to insert entries for all mileage columns at the new age. On success, it shows a success message, closes the add-age modal, resets the age input to 90 months, and reloads the RV matrix to reflect the new row. On error, it logs the error and shows an error message. The finally block always clears the loading state. This ensures the new age row is created across all existing mileage columns for the selected model, maintaining matrix consistency.

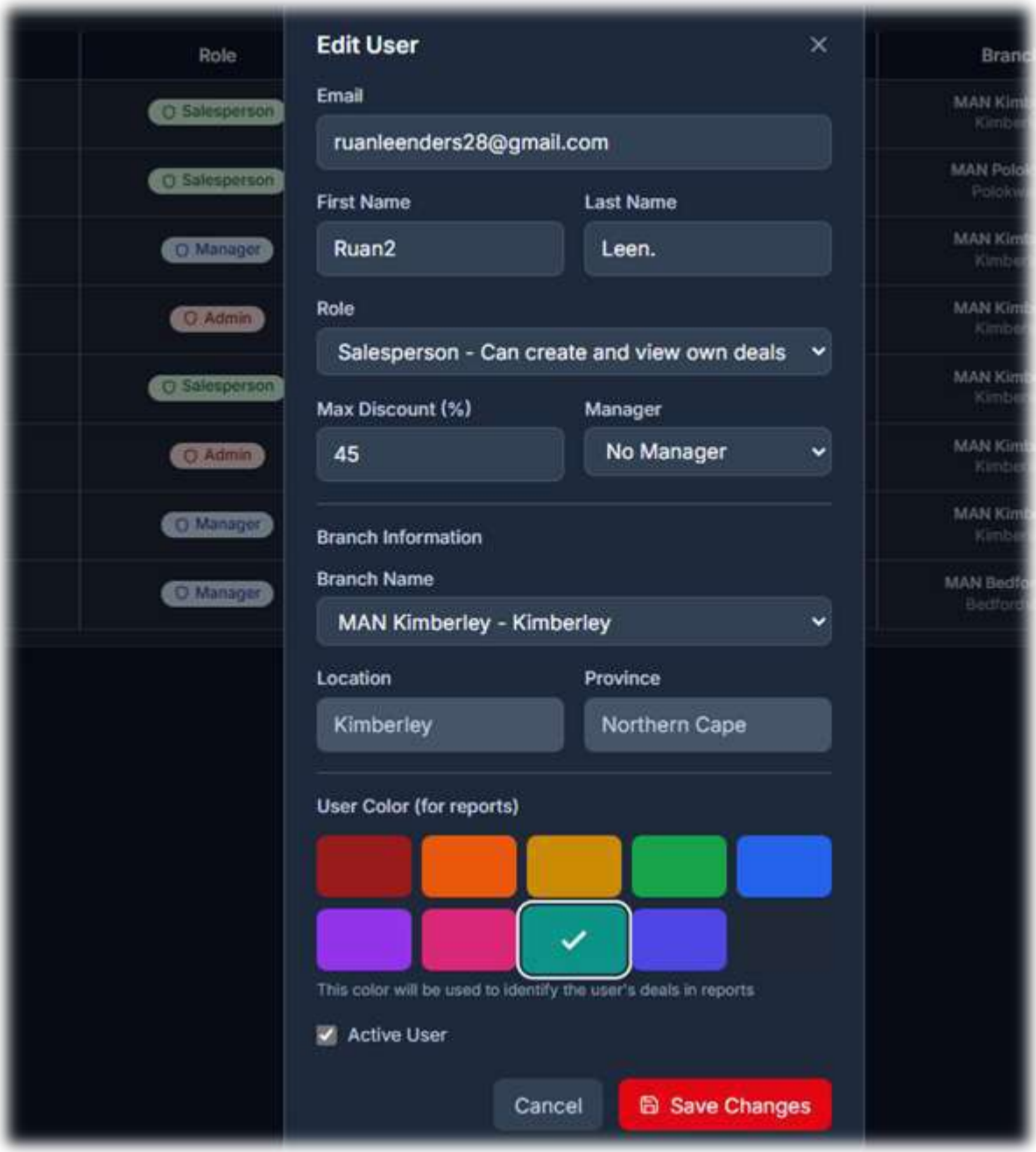
Deal Owner:

TASK 3: Colour coding in User Management to identify owners and their deals.

The colour coding system was added to visually identify deal owners and their deals across the application. It required adding a color column to the users table and integrating colour assignment into the User Management interface. Previously, there was no visual way to distinguish deal owners in reports or user listings, making it difficult to identify which deals belonged to which users at a glance. This was fixed by implementing a comprehensive colour management system that assigns unique colours to each user and displays them consistently across the User Management interface and Deal Reports table. The database implementation required adding a color column to the users table with a VARCHAR(7) data type to store hexadecimal color codes, allowing each user to have a custom colour stored directly in their profile record. Default colours were assigned based on user roles to ensure immediate visual differentiation, with administrators receiving red (#991b1b), managers receiving orange (#ea580c), and salespersons receiving yellow (#eab308). The system includes a predefined palette of ten distinct colours (Red, Orange, Yellow, Green, Blue, Purple, Pink, Teal, Indigo, and Black) that administrators can choose from when creating or editing user profiles, ensuring consistent colour choices while providing sufficient variety for visual identification.

The User Management interface was enhanced with an interactive colour picker that displays all available colour options in a grid layout, allowing administrators to visually select a colour for each user during user creation or profile updates. When a colour is selected, it is immediately saved to the user's profile record in the database, and the selection is visually indicated with a checkmark overlay and ring highlight effect. This colour assignment functionality is integrated seamlessly into both the add user modal and edit user modal, ensuring that colour management is consistently available throughout the user management workflow. The Deal Reports table implementation displays the deal owner's name in a dedicated column with a gradient background that uses the assigned user colour, creating an immediate visual association between deals and their owners. The system implements a sophisticated colour rendering approach that creates a linear gradient from darker shades of the user's colour, providing depth and visual appeal while maintaining readability of the owner's name. A shadow effect is applied using darker variants of the user's colour, further enhancing the visual distinction and creating a three-dimensional appearance that makes each owner's deals stand out clearly in the reports interface. When a deal owner cannot be found or has no assigned colour, the system falls back to a default red colour (#991b1b), ensuring that all deals are visually represented even when user data is incomplete.

The colour coding system enables administrators and managers to quickly identify deal ownership patterns across the entire deal portfolio, supporting visual analysis of sales distribution, workload assessment, and deal tracking. The visual identification extends beyond simple name display, as the consistent colour scheme allows users to rapidly scan deal reports and identify deals belonging to specific team members without reading individual names. This feature significantly enhances the usability of the Deal Reports interface, particularly in scenarios where managers need to review multiple deals simultaneously or assess team performance through visual inspection of deal ownership distribution.



NO	DEAL TITLE	OWNER	CUSTOMER
1	Quote 2_31_Oct	Philip Kalil Zackey	PTG Transport CC
2	Test 31 Oct	Philip Kalil Zackey	PTG Transport CC
3	JvR	Juan Van Rensburg	JvR Transport
4	asdasda	Juan Van Rensburg	Juan Van Rensburg

Showing 1 to 4 of 14 deals

```
<td className="py-2 px-2 border-r border-slate-500 text-xs text-center text-white truncate shadow-lg"
  style={(() => {
    const userId = item.dealCover.userId;
    const user = availableUsers.find(u => u.id === userId);
    const userColor = user?.color || '#991b1b'; // Default red
    console.log('🎨 User Color Debug:', { userId, userName: user?.name, color: user?.color, finalColor: userColor });

    // Helper function to darken color for gradient effect
    const darkenColor = (color: string, percent: number) => {
      const num = parseInt(color.replace('#', ''), 16);
      const amt = Math.round(2.55 * percent);
      const R = Math.max(0, (num >> 16) - amt);
      const G = Math.max(0, ((num >> 8) & 0x00FF) - amt);
      const B = Math.max(0, (num & 0x0000FF) - amt);
      return `#${(0x1000000 + (R * 0x10000) + (G * 0x100) + B).toString(16).slice(1)}`;
    };

    const darkerShade = darkenColor(userColor, 15);
    const shadowColor = darkenColor(userColor, 25);

    return {
      backgroundImage: `linear-gradient(to right, ${darkerShade}, ${userColor}, ${darkerShade})`,
      boxShadow: `0 10px 15px -3px ${shadowColor}80, 0 4px 6px -4px ${shadowColor}80`
    };
  })()}>
  {(() => {
    const userId = item.dealCover.userId;
    const user = availableUsers.find(u => u.id === userId);
    return user ? (user.name || user.email) : (userId ? 'Unknown' : '-');
  })()}
</td>
```

The code renders a table cell for the deal owner with dynamic styling. It finds the user from availableUsers using item.dealCover.userId, gets the user's color (defaults to '#991b1b' if missing), and computes a linear gradient background and shadow. The darkenColor helper parses the hex color, extracts RGB, and darkens each channel by a percentage. It generates darker shades at 15% and 25% for the gradient edges and shadow. The gradient goes from darker edges toward the center (using the original color), creating depth. The shadow uses the darker color at 80% opacity and applies two box-shadow layers for separation. The cell content shows the user's name or email if found, or "Unknown" if the user exists but isn't found, or "-" if there's no userId, ensuring the column always renders. Together, this provides immediate visual identification of deal ownership in the reports interface.

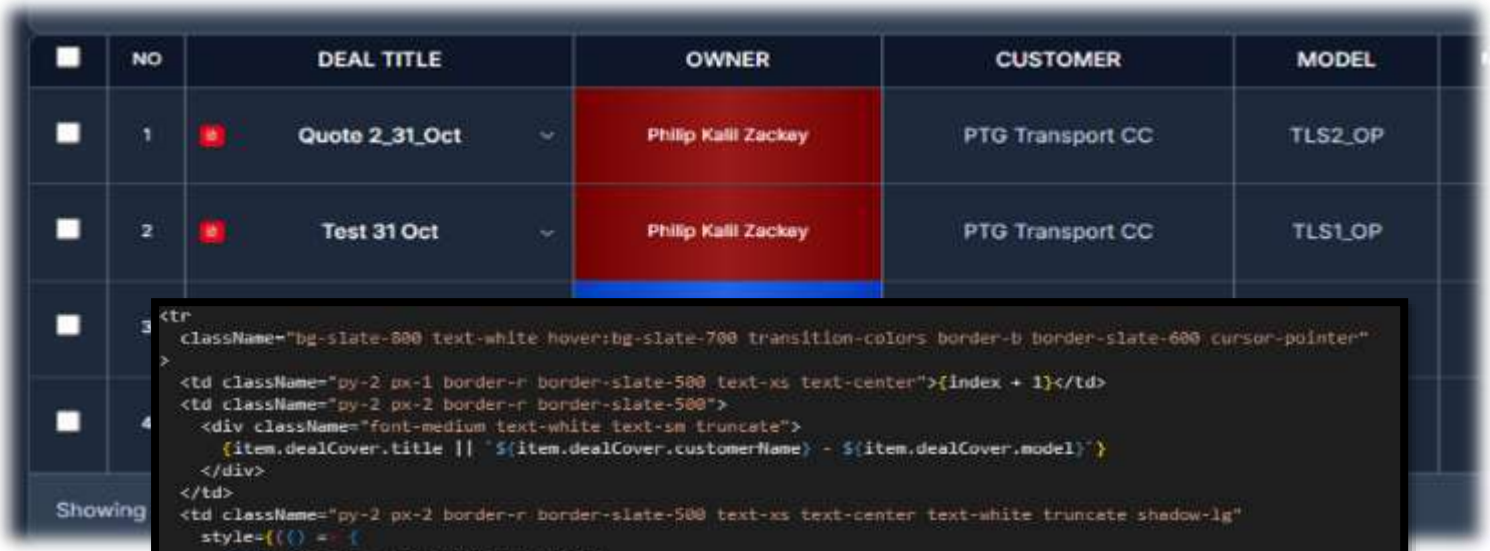
TASK 4: Deal owner column added to Deal Reports table.

A dedicated deal owner column was added to the Deal Reports table to display ownership for each deal. It replaced a hidden userId reference with a visible, color-coded owner display. The column was integrated into the table structure between Deal Title and Customer columns.

The implementation retrieves owner information by matching item.dealCover.userId with the availableUsers list and displays the owner's name or email. It includes fallback handling: if a user isn't found in the list but a userId exists, it shows "Unknown"; if no userId exists, it shows "-". This prevents empty or broken displays when user data is missing.

The column applies dynamic styling using the owner's assigned color, creating a gradient background from darker edge shades to the center color, with a colored shadow. This enables quick visual identification of deals by owner. The column header uses "USER" to indicate ownership, and the display logic handles missing or invalid user references gracefully.

Additionally, the expanded row view shows detailed owner information (name, email, role) when a deal row is expanded. The column integrates with the filtering system, allowing administrators to filter deals by owner. The color-coding system works in conjunction with this column to provide immediate visual association between deals and their owners, improving usability for workload assessment, sales tracking, and deal management across the organization.




```
<tr
  className="bg-slate-800 text-white hover:bg-slate-700 transition-colors border-b border-slate-600 cursor-pointer"
>
  <td className="py-2 px-1 border-r border-slate-500 text-xs text-center">{index + 1}</td>
  <td className="py-2 px-2 border-r border-slate-500">
    <div className="font-medium text-white text-sm truncate">
      {item.dealCover.title || `${item.dealCover.customerName} - ${item.dealCover.model}`}
    </div>
  </td>
  <td className="py-2 px-2 border-r border-slate-500 text-xs text-center text-white truncate shadow-lg"
    style={(() => {
      const userId = item.dealCover.userId;
      const user = availableUsers.find(u => u.id === userId);
      const userColor = user?.color || '#991b1b';

      const darkenColor = (color: string, percent: number) => {
        const num = parseInt(color.replace('#', ''), 16);
        const amt = Math.round(2.55 * percent);
        const R = Math.max(0, (num >> 16) - amt);
        const G = Math.max(0, ((num >> 8) & 0x00FF) - amt);
        const B = Math.max(0, (num & 0x0000FF) - amt);
        return `#${(0x1000000 + (R * 0x10000) + (G * 0x100) + B).toString(16).slice(1)}`;
      };

      const darkerShade = darkenColor(userColor, 15);
      const shadowColor = darkenColor(userColor, 25);

      return {
        backgroundImage: `linear-gradient(to right, ${darkerShade}, ${userColor}, ${darkerShade})`,
        boxShadow: `0 10px 15px -3px ${shadowColor}80, 0 4px 6px -4px ${shadowColor}80`
      };
    })()}>
    {(() => {
      const userId = item.dealCover.userId;
      const user = availableUsers.find(u => u.id === userId);
      return user ? (user.name || user.email) : (userId ? 'Unknown' : '-');
    })()}
  </td>
  <td className="py-2 px-2 border-r border-slate-500 text-sm text-slate-300 truncate text-center">
    {item.dealCover.customerName}
  </td>
  {/* Additional cells... */}
</tr>
```

The code renders a table row for a deal with multiple cells. The row uses Tailwind classes: dark slate background, white text, hover state change, and a bottom border. The first cell shows the row number (index + 1). The second shows the deal title, falling back to customer name and model if title is missing; text truncates to prevent overflow.The third cell is the deal owner column with color coding. An inline style function computes the styling: it extracts userId from the deal cover, looks up the user in availableUsers, and uses the user's color or defaults to red (#991b1b). A darkenColor helper darkens a hex color by a percentage: it parses the hex, extracts RGB via bit shifting, subtracts a computed amount per channel, clamps to non-negative, and reconstructs the hex. Two darker shades are produced: 15% for gradient edges and 25% for the shadow. The cell applies a horizontal gradient from the darker edges to the center color, and a shadow using the 25% darker color at 80% opacity.The cell content uses another inline function: it finds the user by userId and displays the user's name or email if found; if the user isn't found but userId exists, it shows "Unknown"; if there's no userId, it shows "-". This prevents empty or broken displays. The fourth cell displays the customer name with truncation. Together, this creates a color-coded deal owner column that visually identifies ownership while handling missing user data.

TASK 5: Deal owner specified in the TCO Workflow when editing.

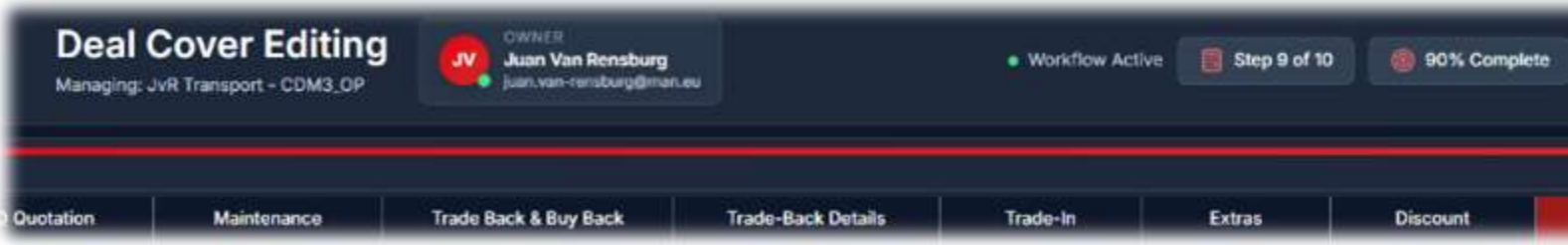
Added a deal owner display in the TCO Workflow interface when editing existing deals. Previously, there was no owner indication during editing, making it unclear who owned the deal.The implementation required adding owner state management and integrating owner information retrieval when loading deals in edit mode.

The system now includes a dealOwner state that stores the owner's first name, last name, and email, which is populated when a deal is loaded for editing. Previously, owner information was stored in the database but not displayed during the editing workflow, making it difficult for users to identify deal ownership while making modifications.When editing a deal, the system extracts owner information from the workflow deal data (workflowDeal.userInfo) and populates the dealOwner state automatically. This ensures that owner information is available throughout the editing session, regardless of which workflow step the user is currently on.

The owner display component is conditionally rendered only when the workflow is in edit mode (isEditMode) and when owner information is available (dealOwner), ensuring that the display does not appear during new deal creation or when owner information is not yet loaded.The visual owner display component presents the owner information in a compact card format within the workflow header section, featuring a circular avatar with the owner's initials displayed in white text on a gradient red background.

The avatar includes a green status indicator badge in the bottom-right corner to signify that the owner is active. The owner's full name is displayed in semibold white text below an uppercase "Owner" label, with the owner's email address shown in smaller gray text beneath the name. This compact design provides immediate visual identification of deal ownership without taking up excessive screen space, maintaining the workflow's focus on deal editing functionality while providing essential ownership context.

The implementation maintains visual consistency with the rest of the TCO Workflow interface, using the same color scheme, border styles, and backdrop blur effects. The owner display is positioned in the header section alongside other workflow status indicators, creating a cohesive visual hierarchy. The component is responsive, hidden on smaller screens (using hidden lg:block) to preserve mobile layout space, while remaining visible on desktop screens where space is more abundant. This responsive approach ensures that the owner information is available when screen real estate allows, while maintaining usability on mobile devices.




```
// Store owner information
if (workflowDeal.userInfo) {
  setDealOwner(workflowDeal.userInfo);
  console.log('✅ Deal owner loaded:', workflowDeal.userInfo);
}
```

This code loads and stores the deal owner information when a deal is opened for editing in the TCO Workflow. It checks whether workflowDeal.userInfo exists before storing it. If it exists, it calls setDealOwner with the owner data (firstName, lastName, email), making it available throughout the editing session. It logs a success message to the console for debugging. This conditional loading ensures owner data is only set when present, avoiding errors if a deal has no owner. By storing it in state, the owner's information remains accessible across workflow steps and can be used for display, validation, or other logic. The code runs when a deal is loaded for editing, ensuring the owner display appears immediately in the workflow interface.

TASK 6: Dropdown display to assign a different owner to an existing deal.

Implemented a dropdown interface within the Deal Reports table that enables administrators and managers to reassign deal ownership from one user to another. The implementation required creating an assignment modal with user selection dropdown and integrating with backend service methods to update deal ownership in the database. Previously, deal ownership could only be changed through direct database access, which slowed deal transfer operations and required technical knowledge. The new interface provides an intuitive way to transfer deals between users, supporting efficient workload management and deal redistribution across sales teams.

The assignment modal is accessible through an "Assign Deal" button located within the expanded deal row view, visible only to users with Admin or Manager roles to ensure appropriate access control. When opened, the modal displays a dropdown containing all users in the system, sorted alphabetically by full name for easy navigation. Each dropdown option displays the user's full name, role, and email address, enabling administrators to quickly identify and select the correct assignee. The dropdown pre-selects the current owner's user ID when the modal opens, making it easy for administrators to verify the existing assignment before making changes. The interface includes a visual indicator section below the dropdown that highlights the current owner's information using red-themed styling, ensuring administrators always know the existing ownership before initiating a transfer.

The assignment process leverages the assignDealToUser service method, which validates that only administrators and managers have permission to perform ownership transfers. The service updates the user_id field in the saved_deal_covers table, maintaining referential integrity with the users table and ensuring data consistency across the application. The implementation includes comprehensive audit logging that records all ownership transfers in the audit logs table, creating a complete audit trail of deal assignment changes for compliance and tracking purposes. After a successful assignment, the deal reports table automatically refreshes to display the updated ownership information, with the new owner's color-coded column immediately reflecting the change, providing instant visual feedback that the assignment was successful.

The user experience includes sophisticated loading states during the assignment process, with a spinner animation and "Assigning..." text displayed on the submit button while the database update is in progress, preventing multiple submissions and providing clear feedback about the operation status. The submit button is disabled until a user is selected from the dropdown, preventing incomplete assignments and ensuring data integrity. Upon successful assignment, a toast notification displays a success message confirming which deal was assigned to which user, providing immediate feedback that the operation completed successfully.


```jsx
{/* User Selection Dropdown */}
<div className="mb-4">
  <label className="block text-sm font-medium text-slate-300 mb-2">
    Select User
  </label>
  <select
    value={selectedUserId}
    onChange={(e) => setSelectedUserId(e.target.value)}
    className="w-full px-3 py-2 bg-slate-700 border border-slate-600 rounded-lg text-white focus:ring-2 focus:ring-red-500 focus:border-red-500"
  >
    <option value="">-- Select a user --</option>
    {allUsers
      .sort((a, b) => {
        const nameA = `${a.firstName} ${a.lastName}`.trim();
        const nameB = `${b.firstName} ${b.lastName}`.trim();
        return nameA.localeCompare(nameB);
      })
      .map((u) => (
        <option key={u.id} value={u.id}>
          {`${u.firstName} ${u.lastName}`.trim()} ({u.role}) - {u.email}
        </option>
      ))}
  </select>
</div>
```

This code implements the user selection dropdown within the deal assignment modal, enabling administrators to choose a new owner for an existing deal. The dropdown component binds to the selectedUserId state variable, which tracks the currently selected user's ID as administrators navigate through the available options. When a user selects an option from the dropdown, the onChange event handler updates the selectedUserId state by calling setSelectedUserId with the selected option's value, ensuring the component reflects the administrator's choice immediately.

The dropdown dynamically populates its options by mapping over the allUsers array, which contains all users in the system retrieved from the database. Before rendering the options, the code sorts the users alphabetically by their full name using the localeCompare method, ensuring that the dropdown presents users in a consistent, easy-to-navigate order regardless of how they are stored in the database. Each dropdown option displays the user's full name (first name and last name combined), their role in parentheses, and their email address, providing administrators with comprehensive information to make informed assignment decisions.

Optimization:

TASK 7: Compressed background image to load faster.

The application's authentication background image was optimized to reduce file size and improve page load performance. The implementation required converting the background image from PNG format to high-quality JPEG format, reducing the file size significantly while maintaining visual quality. Previously, the application used a PNG format background image that was several megabytes in size, causing slow initial page loads, especially on slower network connections or mobile devices. The optimization process involved converting the original PNG image to JPEG format using ImageMagick's convert command with a quality setting of 95, which provides an optimal balance between file size reduction and visual quality preservation.

The code was updated to reference the compressed JPEG file instead of the original PNG file, ensuring that the application loads the smaller file automatically without requiring user intervention. The background image selection logic remained unchanged, using the same theme-based selection mechanism where dark mode displays one background image and light mode displays another. The compressed image maintains the same visual appearance as the original PNG while significantly reducing bandwidth requirements and improving page load times across all device types and network conditions.

The implementation included deployment scripts that automated the conversion process, ensuring consistency across development and production environments. The scripts converted the background image using ImageMagick with quality settings that preserve visual fidelity while achieving maximum file size reduction. After conversion, the application was rebuilt and redeployed with the compressed image, immediately improving performance metrics for all users accessing the authentication pages. The optimization resulted in faster page load times, reduced bandwidth usage, and improved user experience, particularly for users on slower network connections or mobile devices where file size directly impacts loading performance.

```
// Background image based on theme
const backgroundImage = isDarkMode
  ? '/titel_xmas-1.jpg'
  : '/samurai-bg.jpg';

return (
  <div
    key={`unified-auth-${isDarkMode}`}
    className="min-h-screen bg-slate-900 flex items-center justify-center relative"
    style={{
      backgroundImage: `url(${backgroundImage})`,
      backgroundSize: 'cover',
      backgroundPosition: 'center',
      backgroundRepeat: 'no-repeat',
      backgroundAttachment: 'fixed'
    }}
  >
```

This code implements the background image selection and application logic for the authentication form, enabling theme-based background image selection and optimizing load performance through compressed image formats. The code declares a backgroundImage constant that selects between two compressed JPEG files based on the isDarkMode state variable, ensuring that dark mode displays one background image while light mode displays another. The selection uses a ternary operator to conditionally assign the appropriate image path based on the current theme setting, providing a simple and efficient way to switch backgrounds dynamically.

The backgroundImage constant is then used within the div element's inline style object, where it is applied as a CSS background-image property using template literal syntax to construct the URL. The style object includes additional CSS properties that control how the background image is displayed, including backgroundSize set to 'cover' to ensure the image fills the entire container, backgroundPosition set to 'center' to center the image horizontally and vertically, backgroundRepeat set to 'no-repeat' to prevent image tiling, and backgroundAttachment set to 'fixed' to create a parallax scrolling effect where the background remains stationary while content scrolls over it.

The implementation uses compressed JPEG file extensions (.jpg) instead of PNG format, which significantly reduces file size while maintaining visual quality.

TASK 8: Optimized queries for load optimization.

The application's database query performance was significantly improved through batch query optimization, conditional data loading, and parallel query execution. The implementation required refactoring multiple service methods to use batch queries instead of individual queries, reducing database round trips and improving overall load times. Previously, the application executed individual database queries for each record when loading related data, resulting in hundreds of sequential queries that caused slow page loads, especially when displaying large datasets like vehicle models or deal reports. The optimization process involved restructuring queries to fetch related data in parallel using Promise.all(), implementing conditional loading to only fetch data when needed, and using batch queries with IN clauses to retrieve multiple records in a single database round trip.

The Vehicle Management Service was refactored to use batch query optimization, where all vehicle models are first loaded in a single query, then all related data (residual values, service contracts, specifications, and pricing history) are fetched in parallel using Promise.all(). The implementation creates lookup maps (Map objects) for fast data access, eliminating the need for nested loops or repeated queries when associating vehicles with their related data. This approach reduces query execution time from hundreds of individual queries to just a few batch queries executed in parallel, dramatically improving load performance for vehicle management interfaces.

The Deal Reports component received similar optimization through conditional batch loading, where trade-in summaries and TCO reports are only loaded when needed based on component filters. The implementation uses batch loading with Promise.all() to fetch all trade-ins or TCO reports for multiple deals simultaneously, rather than loading them individually for each deal. The component tracks loading performance with timing metrics and provides console logging to monitor optimization effectiveness. Error handling was added to individual batch query operations to ensure that failures in one query don't prevent other queries from completing, improving overall system reliability.

The optimization resulted in measurable performance improvements, with load times reduced from several seconds to milliseconds for large datasets. The batch query approach scales better as data volumes increase, since the number of database round trips remains constant regardless of record count, whereas the previous approach increased queries linearly with record count. The implementation maintains data consistency and accuracy while providing significant performance gains, ensuring that the application remains responsive even when handling thousands of records across multiple related tables.

```
// Batch fetch all integration data in parallel
const [
  rvData,
  serviceContractsData,
  specificationsData,
  pricingStatsData
] = await Promise.all([
  // Get all RV data in one query
  supabase
    .from('residual_value_tables')
    .select('model_code, id')
    .in('model_code', vehicles.map(v => v.code)),

  // Get all service contracts data
  supabase
    .from('service_contracts')
    .select('vehicle_model, id')
    .in('vehicle_model', vehicles.map(v => v.code)),

  // Get all specifications data
  supabase
    .from('vehicle_specifications')
    .select('vehicle_id, id')
    .in('vehicle_id', vehicles.map(v => v.id)),

  // Get all pricing history stats
  supabase
    .from('vehicle_pricing_history')
    .select('vehicle_model_id, change_date')
    .in('vehicle_model_id', vehicles.map(v => v.id))
    .order('change_date', { ascending: false })
]);

// Create lookup maps for fast access
const rvMap = new Map();
rvData.data?.forEach(rv => {
  if (!rvMap.has(rv.model_code)) rvMap.set(rv.model_code, []);
  rvMap.get(rv.model_code).push(rv);
});
```

This code implements parallel batch query execution for loading related vehicle data, optimizing database performance by executing multiple queries simultaneously instead of sequentially. The code uses Promise.all() to execute four separate database queries in parallel, each fetching different types of related data (residual values, service contracts, specifications, and pricing history) using batch queries with IN clauses. The Promise.all() method ensures that all queries execute concurrently, reducing total query time from the sum of individual query times to the time of the slowest query, which is typically much faster than sequential execution.

Each query uses the .in() method to fetch multiple records in a single database round trip, passing an array of vehicle codes or IDs that was extracted from the previously loaded vehicles array using the map() method. This batch query approach eliminates the need for individual queries for each vehicle, reducing database round trips from potentially hundreds of queries to just four parallel queries, regardless of the number of vehicles in the dataset. The queries are structured to only select the fields needed for data association (model_code, vehicle_model, vehicle_id, etc.), minimizing data transfer and improving query performance.

Services:

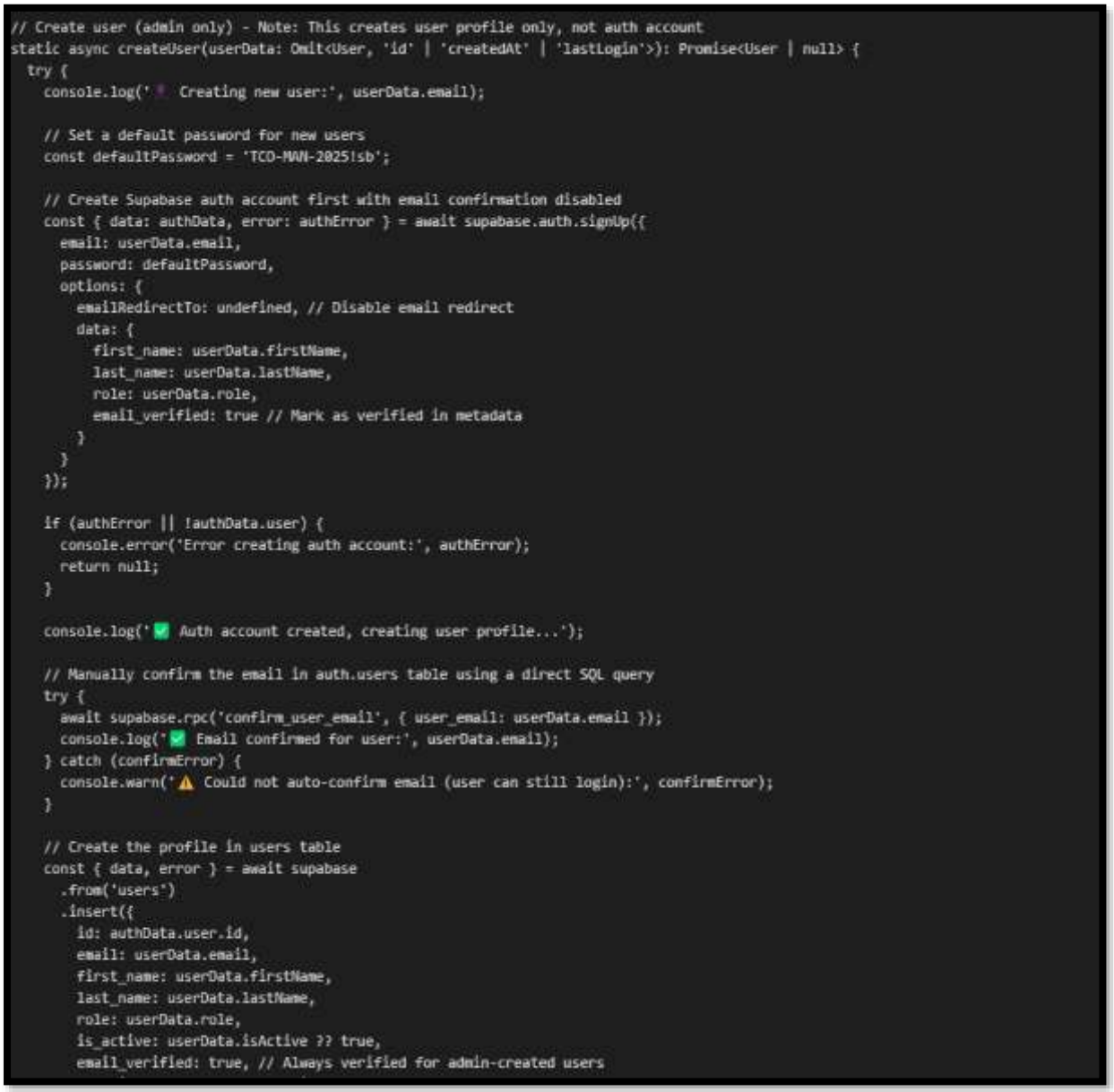
TASK 9: Password reset fixed, and signup forum removed (Add user in user management their default password is TCO-M*N-2025!sb, they can then reset it).

The authentication system was restructured to centralize user creation through User Management, removing public signup functionality and implementing a robust password reset system. The implementation required removing the signup form from public routes, fixing password reset functionality to use a verification code system, and ensuring new users created through User Management receive a secure default password. Previously, users could self-register through a public signup form, which created administrative overhead and potential security concerns. The new approach requires administrators to create user accounts through the User Management interface, ensuring proper role assignment and account verification before users can access the system.

When administrators create new users through User Management, each user receives a standardized default password (TCO-MAN-2025!sb), which they must change upon first login using the password reset functionality. The user creation process automatically creates both the authentication account and the user profile in the users table, with email verification automatically confirmed to eliminate email confirmation delays. The system creates the Supabase auth account with the default password, then confirms the email using a database function (confirm_user_email), ensuring that newly created users can immediately log in with their default credentials without waiting for email confirmation.

The password reset functionality was fixed to use a verification code system instead of password reset links, improving security and user experience. When users request a password reset, the system generates a verification code using a database function (create_password_reset_token) and stores it in the database with expiration tracking. The verification code is sent to the user's email address, and users must enter this code along with their new password to complete the reset process. The system validates that users exist and are active before generating codes, prevents code reuse after successful resets, and includes expiration logic to invalidate codes after 15 minutes for security purposes.

The password reset process includes comprehensive validation, checking that passwords match, meet minimum length requirements (8 characters), and that verification codes are valid and not expired. The implementation uses a dedicated password reset service endpoint that securely handles password updates, ensuring that passwords are properly hashed and stored in the Supabase authentication system. Error handling provides clear feedback to users if codes are invalid, expired, or if other errors occur during the reset process. The removal of public signup ensures that only authorized administrators can create accounts, reducing the risk of unauthorized access and simplifying user onboarding through a controlled, managed process.



```
// Create user (admin only) - Note: This creates user profile only, not auth account
static async createUser(userData: Omit<User, 'id' | 'createdAt' | 'lastLogin'>): Promise<User | null> {
  try {
    console.log('🔄 Creating new user:', userData.email);

    // Set a default password for new users
    const defaultPassword = 'TCO-MAN-2025!sb';

    // Create Supabase auth account first with email confirmation disabled
    const { data: authData, error: authError } = await supabase.auth.signUp({
      email: userData.email,
      password: defaultPassword,
      options: {
        emailRedirectTo: undefined, // Disable email redirect
        data: {
          first_name: userData.firstName,
          last_name: userData.lastName,
          role: userData.role,
          email_verified: true // Mark as verified in metadata
        }
      }
    });

    if (authError || !authData.user) {
      console.error('Error creating auth account:', authError);
      return null;
    }

    console.log('✅ Auth account created, creating user profile...');

    // Manually confirm the email in auth.users table using a direct SQL query
    try {
      await supabase.rpc('confirm_user_email', { user_email: userData.email });
      console.log('✅ Email confirmed for user:', userData.email);
    } catch (confirmError) {
      console.warn('⚠️ Could not auto-confirm email (user can still login):', confirmError);
    }

    // Create the profile in users table
    const { data, error } = await supabase
      .from('users')
      .insert({
        id: authData.user.id,
        email: userData.email,
        first_name: userData.firstName,
        last_name: userData.lastName,
        role: userData.role,
        is_active: userData.isActive ?? true,
        email_verified: true, // Always verified for admin-created users
```

This code implements the user creation functionality for administrators, centralizing user account creation through User Management and eliminating the need for public signup forms. The function creates both the authentication account and user profile in a single operation, ensuring data consistency between the Supabase authentication system and the application's user database. The implementation sets a standardized default password (TCO-MAN-2025!sb) for all newly created users, which they must change upon first login using the password reset functionality.

The function first creates the Supabase authentication account using the signUp method, passing the user's email and the default password. The signup options disable email redirects and mark the email as verified in metadata, eliminating email confirmation delays that would prevent users from logging in immediately. After the authentication account is created, the code calls a database function (confirm_user_email) to manually confirm the email address in the auth.users table, ensuring that users can log in without waiting for email verification links.

TASK 10: PDF generation of mapping fixed (Generate Quote).

PDF Generation of Mapping Fixed (Generate Quote):

The quote generation PDF mapping system was fixed to ensure accurate field mapping between the frontend quote data and the Word template placeholders, resolving mapping inconsistencies that caused missing or incorrectly positioned data in generated quotes. The implementation required comprehensive mapping updates to match exact placeholder names from the frontend with the Word template fields, ensuring that all quote data including customer information, vehicle details, financial parameters, trade-ins, and extras are correctly mapped and displayed in the generated PDF documents.

Previously, the quote generation process had mapping inconsistencies where some template placeholders didn't match the data being passed from the frontend, resulting in missing data, incorrect field positioning, or empty fields in generated quotes. The fix involved creating a comprehensive mapping structure that handles multiple naming variations (camelCase, PascalCase, snake_case) to ensure compatibility with different template formats, while maintaining backward compatibility with existing quote structures. The implementation maps all customer fields (CustomerName, Customer, CustomerEmail, Phone, Address, Contact), quote metadata (Quote, Date), vehicle information (VehicleModel, Quantity, UnitPriceExclVAT), financial data (TOTAL, SubtotalExcl, Discount, PrincipleDebt, InterestPercent, TotalDeposit), and dynamic tables (trade-ins, extras) correctly to their respective template placeholders.

The mapping system now handles complex nested data structures, including legacy quote data formats and workflow-based quote data, ensuring that quotes generated from both traditional deal covers and TCO workflow deals are correctly processed. The implementation includes proper data type conversion, parsing numeric values to fixed decimal places for currency fields, and formatting date strings correctly for template display. The system maps sales executive information including name, email, manager details, and manager email, with multiple case variations (salesExecutive, SalesExecutive) to ensure template compatibility regardless of placeholder naming conventions used in different Word templates.

The fix includes comprehensive error handling to provide fallback values for missing data, ensuring that generated quotes never contain undefined or null values that would break template rendering. The mapping logic handles array-based data (trade-ins, extras) by formatting them into both loop-compatible arrays and pre-formatted text tables, supporting different template structures. The implementation maintains consistency across all quote generation endpoints, ensuring that quotes generated from deal management, workflow steps, and report interfaces all use the same mapping logic, eliminating discrepancies between different quote generation sources.

Quote: {Quote}
Customer: {Customer}
Date: {Date}

Quotation for: MAN TGS 26.440 6X4 BL SA TM

Dear {CustomerName}

We thank you for your enquiry for the MAN TGS 26.440 6X4 BL SA TM and take pleasure in submitting herewith our quotation for the supply and delivery of the products, as detailed below.

1. **Vehicle selection**

Please refer to the Scope of Supply (below). Should you require any additional information, please feel free to contact us.

Based on the specification the pricing of our offering is as follows.

MODEL	QTY	TOTAL
MAN TGS 26.440 6X4 BL SA TM	{QTY}	R{TOTAL}

2. **Delivery**

The capacity for us to deliver the products offered is determined by the following factors:

- Final quantity of trucks required
- Agreement of final specification and execution
- Finalisation of the commercial terms of the order
- Engineering, testing, documentation, homologation and sign-off (if required)

A final delivery schedule can be agreed at the time of order confirmation and finalisation, subject to stock availability and prior sale.

3. **Payment Terms**

Our normal terms are payment in full before delivery of the complete product to your operation. Any variations to these terms of payment are to be agreed between us and confirmed in writing.

4. **Validity**

The prices quoted above are valid for acceptance for a period of 14 days from the date of this quotation and will be subject to review thereafter.

5. **Warranty**

{warrantyDescription}

6. **Customer requested extras**

Code	Description	Supplier	Cost	Selling Price
{#extras}{code}	{description}	{supplier}	{cost}	{selling price excl vat}{/extras}

7. **Repair and Maintenances**

Based on a monthly utilization of {monthlyMileage} per month, and annual mileage of {annualMileage} with a total distance of {totalMileage} – MAN can offer a {maintenanceType} contract rate of R{maintenanceCost}.

Subject to MAN Repair and Maintenance terms and conditions, and your acceptance thereof.

8. **Trade Back**

Based on a total distance of {tradeBackPeriod} and/or {Mileage}. MAN is able to offer a trade back value of R{TradeBackValue}.

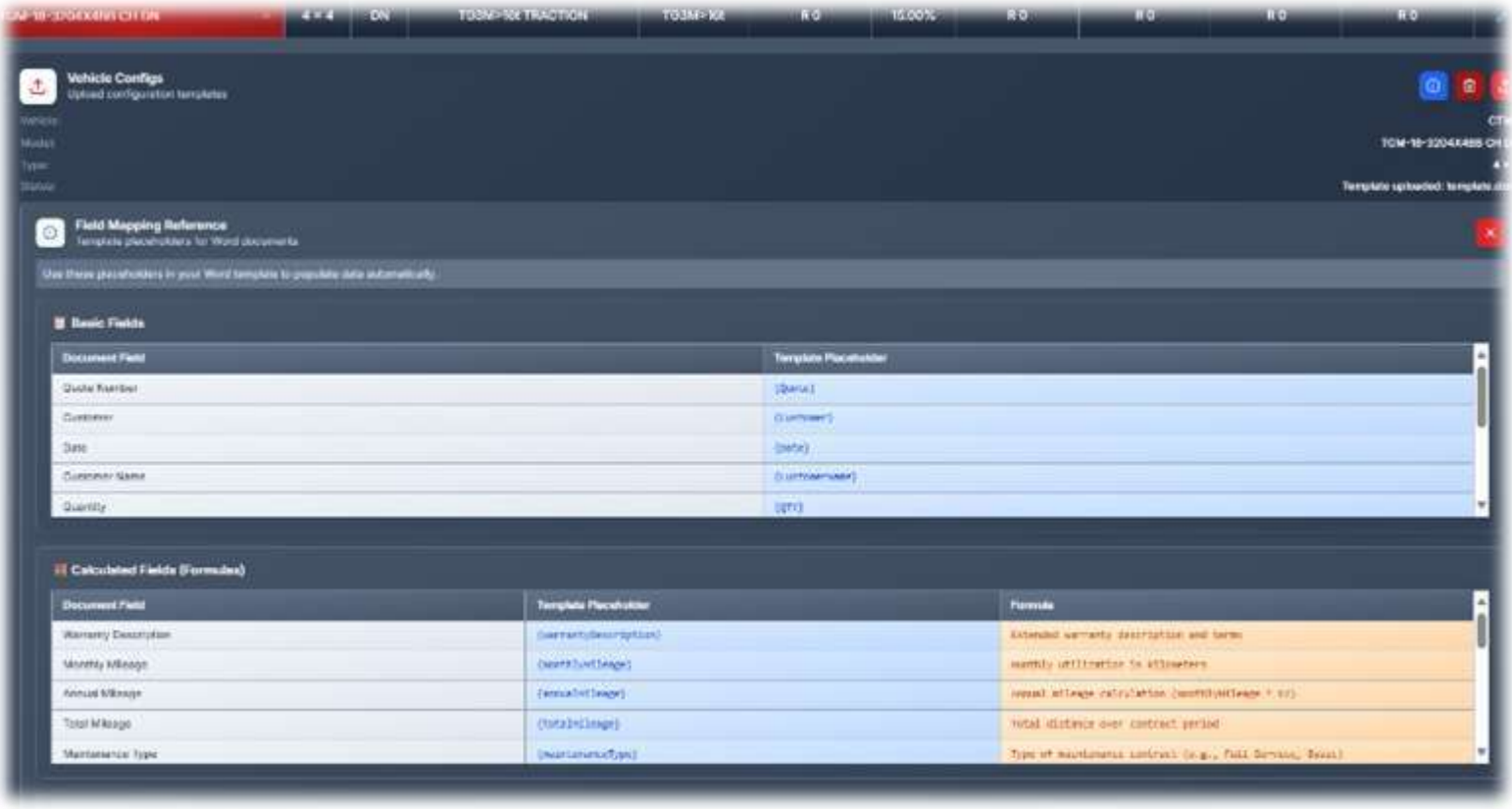
Subject to MAN Top Used terms and conditions, and your acceptance thereof.

9. **Trade In**

Our team will assess the condition and market value of your current vehicle/s to offer you a fair and competitive trade-in amount. The offer can be used directly towards your new vehicle/s, making the upgrade process simple, transparent, and cost-effective. This ensures a smooth transition while helping you get the best possible value for your existing asset.

Model	Year Model	Mileage	Offer
{#tradeins}{model}	{year}	{mileage}	{offer}{/tradeins}

All trade-in valuations are subject to final inspection and approval. The final offer may vary based on the vehicle's actual condition, mileage, and market factors at the time of assessment. Subject to Top Used T and Cs in terms and conditions.



```
// Prepare data for template - use the exact data structure from frontend
const templateData = {
  // Match the exact placeholder names from the frontend
  CustomerName: quote.CustomerName || '',
  Customer: quote.Customer || '',
  CustomerEmail: quote.CustomerEmail || '',
  Phone: quote.Phone || '',
  Address: quote.Address || '',
  Contact: quote.Contact || '',

  Quote: quote.Quote || '',
  Date: quote.Date || new Date().toLocaleDateString(),

  VehicleModel: quote.VehicleModel || '',
  Quantity: quote.Quantity || 1,
  UnitPriceExclVAT: parseFloat((quote.UnitPriceExclVAT || 0).toFixed(2)),

  TOTAL: parseFloat((quote.TOTAL || 0).toFixed(2)),
  SubtotalExcl: parseFloat((quote.SubtotalExcl || 0).toFixed(2)),
  Discount: parseFloat((quote.Discount || 0).toFixed(2)),

  // Financial placeholders from your template
  PrincipleDebt: parseFloat((quote.PrincipleDebt || 0).toFixed(2)),
  InterestPercent: quote.InterestPercent || '12.5',
  InterestType: quote.InterestType || 'Fixed',
  TotalDeposit: parseFloat((quote.TotalDeposit || 0).toFixed(2)),
  FinancialPeriod: quote.FinancialPeriod || '60 months',
  Mileage: quote.Mileage || '100,000 km',
  TradeBackValue: parseFloat((quote.TradeBackValue || 0).toFixed(2)),
  MonthlyExcessKmCharge: quote.MonthlyExcessKmCharge || 'R2.50',
  TotalInstalment: parseFloat((quote.TotalInstalment || 0).toFixed(2)),

  // VAT and pricing
  UnitVAT: parseFloat((quote.UnitVAT || 0).toFixed(2)),
  VAT15: parseFloat((quote.VAT15 || 0).toFixed(2)),
  DeliveryFee: parseFloat((quote.DeliveryFee || 5000).toFixed(2)),
  CashDeposit: parseFloat((quote.CashDeposit || 0).toFixed(2)),

  // Dynamic trade-ins table (multiple formats)
  tradeIns: formatTradeInsForTable(quote.tradeInData || []),
  tradeInsTable: generateTradeInsTextTable(quote.tradeInData || []),
  hasTradeIns: (quote.tradeInData || []).length > 0,

  // Dynamic extras table (array for looping)
  extras: formatExtrasForTable(quote.extrasData || []),
  hasExtras: (quote.extrasData || []).length > 0,
```

This code implements comprehensive field mapping for quote generation, ensuring that all quote data from the frontend is correctly mapped to Word template placeholders with proper formatting and type conversion. The mapping structure creates a templateData object that contains all fields required for quote generation, using exact placeholder names that match the Word template's field names, ensuring that Docxtemplater can correctly replace placeholders with actual data during PDF generation.

The mapping handles multiple data sources and naming conventions, providing fallback values using the logical OR operator (||) to ensure that missing data doesn't break template rendering. For example, customer name mapping checks both `quote.CustomerName` and `quote.Customer` fields, using an empty string as a fallback if neither exists. Numeric fields are parsed and formatted to fixed decimal places (2 decimals) using `parseFloat()` and `toFixed()`, ensuring consistent currency formatting in generated quotes, while preventing floating-point precision errors that could cause incorrect totals.

MAN SPR

2.1.1 About

The SPR (Special Price Request) System is a comprehensive, role-based workflow management platform designed to streamline and automate the entire lifecycle of special pricing requests for spare parts in the automotive/manufacturing industry. Built on ASP.NET Core (Blazor Server) with advanced workflow orchestration, it delivers a fully integrated solution for dealer request submission, vetting, approval, document management, and credit note processing.

The system transforms the traditional manual process of handling special price requests into a digital, auditable, and efficient workflow that ensures proper authorization, document validation, and financial tracking at every stage.

2.1.2 Market Opportunity

The global automotive aftermarket parts market is projected to reach $513.8 billion by 2030, with the spare parts pricing and discount management segment showing significant growth. The SPR System addresses the critical need for mid-to-large automotive dealerships and parts distributors to:

- **Automate Price Approval Workflows**: Eliminate manual paperwork and email-based approval processes
- **Ensure Compliance**: Maintain audit trails and digital signatures for financial accountability
- **Improve Turnaround Time**: Reduce processing time from days to hours through automated routing
- **Enhance Visibility**: Provide real-time dashboards and reporting for all stakeholders
- **Manage Pricing Data**: Centralize spare parts pricing reference data with bulk upload capabilities

The system targets automotive dealerships, parts distributors, and manufacturing organizations that require structured approval processes for special pricing requests, avoiding the complexity and high costs associated with enterprise ERP pricing modules.

2.1.3 Technical Architecture

- **Built on ASP.NET Core 8.0** with Blazor Server for real-time, interactive web applications
- **Entity Framework Core** with SQL Server for robust data persistence and relationships
- **Role-Based Access Control (RBAC)** using ASP.NET Core Identity with custom roles (SPRUser, SPRApprover, SPRSuperUser)
- **Workflow Orchestration Engine** managing 7-stage approval process with state machine pattern
- **Document Management System** with secure file upload, storage, and retrieval

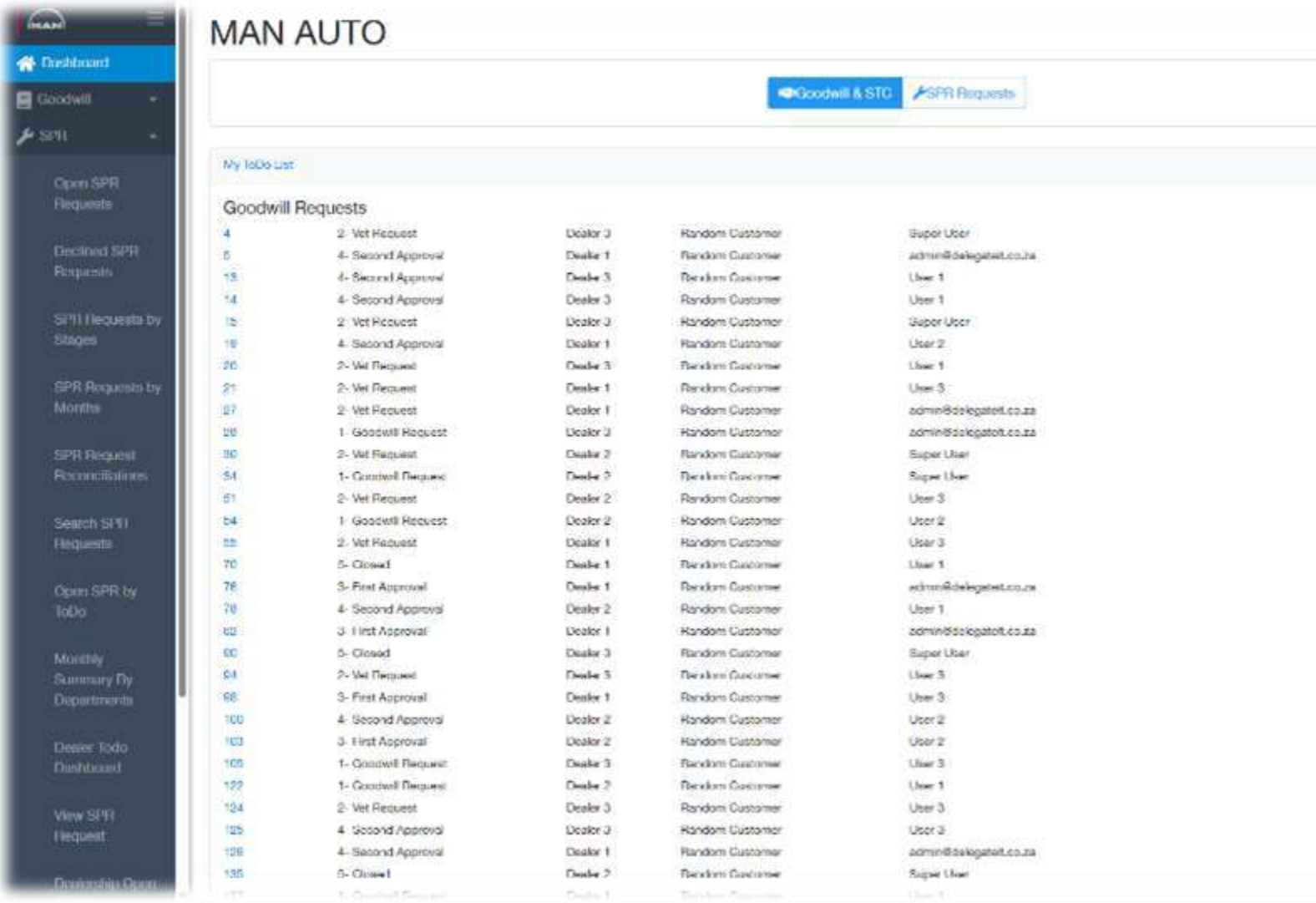
- **Real-time Notifications** system for email and dashboard alerts
- **Audit Trail System** maintaining complete history of all actions and changes

2.1.4 Technology Stack

- **ASP.NET Core 8.0**: Modern web framework
- **Blazor Server**: Real-time interactive UI
- **Entity Framework Core**: ORM for database operations
- **SQL Server**: Database management system
- **ASP.NET Core Identity**: Authentication and authorization
- **Dependency Injection**: IoC container for service management

2.1.5 Conclusion

The SPR (Special Price Request) System is a comprehensive, production-ready solution that transforms manual pricing approval processes into an efficient, automated, and compliant digital workflow. With its robust architecture, extensive feature set, and focus on user experience, it provides significant value to automotive dealerships and parts distributors seeking to modernize their special pricing operations.



2.1.6 Summary

TASK 1: Developed a spare parts pricing upload feature under the Settings menu accessible via /sprpriceupload, restricted to users with the SPRSuperUser role using [Authorize(Roles = "SPRSuperUser,Admin")].

TASK 2: Implemented CSV and Excel file upload using ClosedXML for Excel processing and StreamReader for CSV parsing, supporting bulk pricing data uploads.

TASK 3: Created validation logic in SPRPriceDataService.cs that checks file headers for required columns (PartNumber, RetailPrice, DealerPrice) before processing uploads.

TASK 4: Built an update system in SPRPriceDataService.cs that compares new pricing data with existing records using GetByPartNumberAsync() and only updates when prices have changed.

TASK 5: Added an updated_at timestamp field to the SPRPriceData model that tracks when each pricing row was last modified using DateTime.Now.

TASK 6: Implemented success and failure feedback displays in SPRPriceUpload.razor showing the number of updated rows versus skipped rows in the upload results.

TASK 7: Created the SPR submission form in NewSPRRequest.razor that allows dealers to submit multiple spare parts in a single request using SfGrid component.

TASK 8: Implemented auto-population in SPRPartItemService.cs using FetchPricesForPartNumberAsync() that populates part descriptions, dealer prices, and retail prices when part numbers are entered.

TASK 9: Built real-time calculation functionality in CalculateTotals() method that computes line totals, subtotals, and savings automatically using JavaScript and C# calculations.

TASK 10: Developed the reference number generation system in SPRWorkflowService.cs using GenerateReferenceNumberAsync() that creates unique SPR-YYYY-NNNN identifiers.

TASK 11: Created dealer document upload functionality in NewSPRRequest.razor using SfUploader component supporting PDF and image files for the submission stage.

TASK 12: Implemented the digital confirmation checkbox requirement in SPRRequest.IsConfirmed property that must be checked before SPR submission using @bind-Value binding.

TASK 13: Built the Vetter Review dashboard in SPRDashboard.razor with a Todo List using GetMyToDoAsync() method showing all submitted SPRs requiring action.

TASK 14: Developed the approved price input functionality in ReviewSPRRequest.razor using SfNumericTextBox that allows vetters to set approved prices per spare item in the SpecialPriceGranted field.

TASK 15: Implemented system-calculated totals in CalculateTotals() method comparing requested versus approved amounts for vetters using RequestedLineTotal and approved price calculations.

TASK 16: Created action buttons in ReviewSPRRequest.razor using SfButton components including Approve, Reject, and Request More Info options that call workflow service methods.

TASK 17: Implemented the workflow in SPRWorkflowService.VetterApproveAsync() that routes approved requests to dealers for document upload at Stage 3 by setting StageId = 3.

TASK 18: Developed the Dealer Todo Dashboard in DealerTodoDashboard.razor using GetDealerToDoAsync() method that displays all SPR requests requiring dealer action.

TASK 19: Created the Dealer Document Upload page in DealerDocumentUpload.razor showing SPR details including approved prices from vetters using GetPartItemsByRequestIdAsync().

TASK 20: Implemented invoice document upload with drag-and-drop in DealerDocumentUpload.razor using SfUploader component supporting PDF, PNG, JPG, and JPEG formats with 10MB file size limit.

TASK 21: Built validation in CanCompleteUpload() method that prevents submission without at least one invoice document attached by checking invoiceDocuments.Any().

TASK 22: Developed the document submission functionality in CompleteUpload() method that calls

MoveToCreditNoteProcessingAsync() moving requests to Stage 4 and notifies vetters.

TASK 23: Created the Vetter Credit Review interface in ReviewSPRRequest.razor allowing vetters to view and download uploaded documents using SPRRequestDocumentService.GetByRequestAsync().

TASK 24: Implemented document validation workflow in SPRWorkflowService.VetterCreditReviewApproveAsync() and VetterCreditReviewRejectAsync() where vetters can approve or reject documents.

TASK 25: Built the Approver Review interface in ViewSPRRequest.razor for final approval or rejection decisions showing all request details, documents, and pricing information.

TASK 26: Developed the final approval workflow in SPRWorkflowService.Approver1ApproveAsync() that moves approved requests to Stage 7 by setting StageId = 7 for credit note upload.

TASK 27: Implemented the rejection workflow in SPRWorkflowService.ApproverRejectAsync() that sends rejected requests to Stage 6 by setting StageId = 6 for vetter final review.

TASK 28: Created the Vetter Final Review functionality in SPRWorkflowService with methods like VetterFinalReviewRequestDocumentsAsync() and VetterFinalReviewResubmitAsync() handling approver rejections.

TASK 29: Developed the Credit Note Upload page in SPRCreditNoteEntry.razor for entering actual quantities and special prices using GetCreditNoteItemsAsync() method.

TASK 30: Implemented the credit note data entry grid in SPRCreditNoteEntry.razor using SfGrid with editable fields and validation for actual quantities and special prices.

TASK 31: Built the credit note calculation engine in CreditNoteCalculationService.cs that computes line totals, VAT at 15%, and grand totals automatically using CalculateCreditNoteTotalsAsync().

TASK 32: Created the credit note document upload functionality in SPRCreditNoteUpload.razor using SfUploader for final document submission supporting PDF format.

TASK 33: Developed validation logic in SPRPartItemService.ValidatePartItemAsync() that skips requested price validation during credit note processing at Stage 7 using skipRequestedPriceValidation: true parameter.

TASK 34: Implemented the workflow orchestration engine in SPRWorkflowService.cs managing all seven stages of the SPR process using state machine pattern with StageId transitions.

TASK 35: Built notification systems using INotificationService that send emails and dashboard alerts for stage transitions, document uploads, approvals, and rejections.

TASK 36: Created comprehensive audit trail functionality in SPRRequestLogService.cs logging all actions with timestamps and user information using CreateLogAsync() method.

TASK 37: Developed the dashboard system in SPRDashboard.razor with Todo List using GetMyToDoAsync(), Watch List using GetMyWatchListAsync(), Approved, and Declined sections.
TASK 38: Implemented search and filtering functionality in SPRRequestService.SearchAsync() for finding SPR requests by reference number, customer name, dealer, or branch criteria.

TASK 39: Built the price data management system in SPRPriceDataService.cs maintaining a master database of spare parts pricing with CRUD operations using Entity Framework Core.

TASK 40: Created role-based access control using ASP.NET Core Identity restricting features based on SPRUser, SPRApprover, and SPRSuperUser roles with [Authorize] attributes.

TASK 41: Implemented multi-vetter support in SPRApprover model allowing multiple vetters without branch restrictions by setting BranchId = null in the database.

TASK 42: Developed the request rejection workflow in SPRWorkflowService with feedback loops between approvers and vetters using rejection reason fields and stage transitions.

TASK 43: Created cancellation functionality in SPRWorkflowService.CancelRequestAsync() allowing vetters and approvers to cancel requests at various stages with cancellation reasons.

TASK 44: Implemented the completion workflow in SPRWorkflowService.CompleteCreditNoteAsync() that finalizes SPR requests when credit notes are processed by setting StateId = 6 and StageId = 8.

TASK 45: Built the document management system in SPRRequestDocumentService.cs that securely stores and tracks all uploaded documents using file system storage in wwwroot/documents/spr/ directory.

TASK 46: Developed the pricing validation system in SPRPartItemService.ValidatePartItemAsync() ensuring requested prices do not exceed dealer or retail prices during submission using HasPriceData checks.

TASK 47: Created the auto-suggestion feature in SPRPartItemService.SearchPartsAsync() for part numbers with automatic price population using LINQ queries on the pricing database.

TASK 48: Implemented the savings calculation system in SPRPartItemViewModel.Savings property comparing requested prices to dealer and approved prices using calculated properties.

TASK 49: Built the discount percentage calculation in SPRPartItemViewModel.DiscountPercentage showing percentage discounts from dealer prices using formula ((DealerPrice - SpecialPrice) / DealerPrice) * 100.

TASK 50: Developed the VAT calculation system in CreditNoteCalculationService.cs computing VAT at 15% on subtotals using VATAmount = SubtotalExclVAT * VATRate formula.

TASK 51: Created the line total calculation system in SPRPartItemViewModel.LineTotal that multiplies quantities by prices in real-time using ActualQuantity * SpecialPriceGranted.

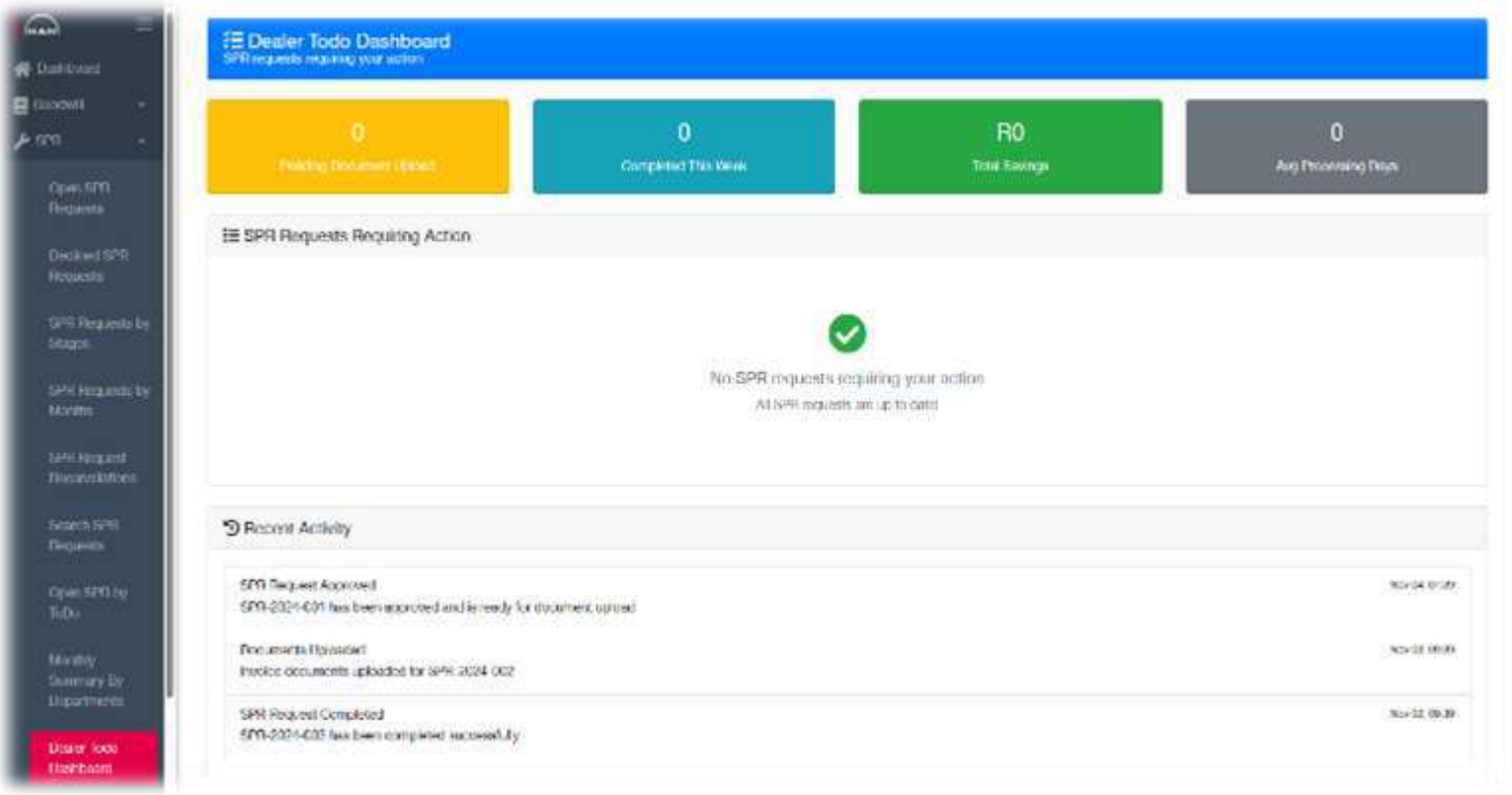
TASK 52: Implemented request summary displays in SPRDashboard.razor showing totals, quantities, and financial summaries using aggregated data from SPRPartItems collection.

TASK 53: Built workflow state management in SPRRequest model tracking requests through all stages from submission to completion using StageId and StateId properties.

TASK 54: Developed error handling and validation feedback in all Razor pages using SfToast components providing clear error and success messages with ShowToast() method.

TASK 55: Created the file processing system in SPRPriceUpload.razor handling CSV and Excel formats using ProcessCsvFile() and ProcessExcelFile() methods for pricing data uploads.

TASK 56: Implemented data update logic in SPRPriceDataService.UpdateAsync() that only modifies records where prices have changed by comparing RetailPrice and DealerPrice values before updating.



PROJECT 1: SPR Multi-Approver Workflow Implementation Documentation

Project Overview

This implementation for this month enhances the Special Price Request (SPR) system for MAN Trucks and Bus, adding multi-approver workflows for price approvals. The SPR system manages dealer requests for special pricing on vehicle parts and components, routing requests through vetters and approvers based on price ranges and business rules. The system supports:

- Dealers submitting SPR requests with part items and proposed prices
- Vetters reviewing and setting approved prices
- Approvers reviewing and approving requests based on price ranges
- Multi-stage workflow from submission to credit note processing
- Document management and audit trails

This update adds multi-approver support, allowing multiple approvers to be assigned to a single SPR request based on custom price ranges, with all approvers required to approve before the request proceeds to the credit note stage.

TASK 1: Multi-Approver Database Schema and Model Updates

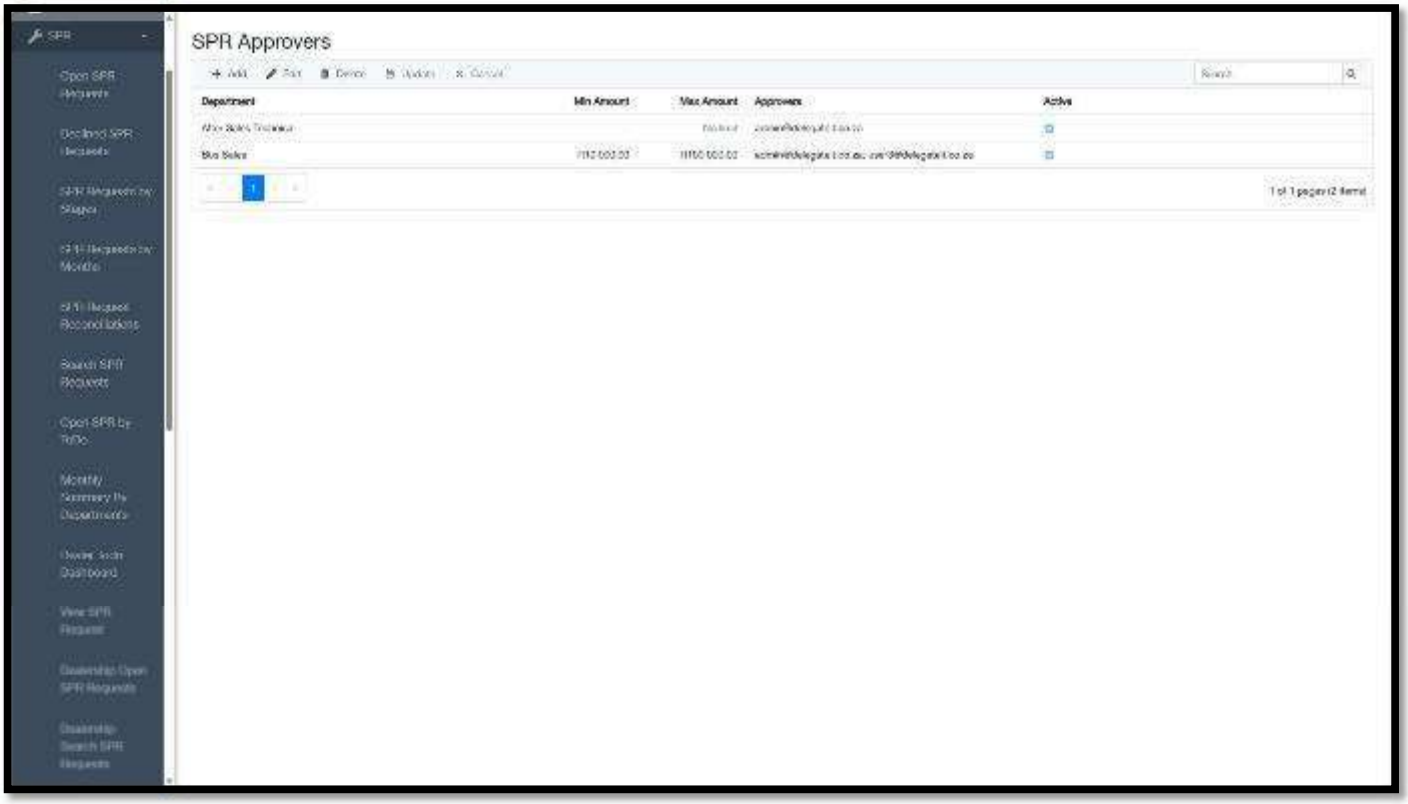
Added database fields to track multiple approvers per SPR request. Previously, only a single approver (Approver1Id) was supported, which blocked assigning multiple approvers to a price range.
Database Changes:

- Added AssignedApproverIds (VARCHAR(500)) to store semicolon-separated IDs of all assigned approvers.
- Added ApprovedApproverIds (VARCHAR(500)) to track which approvers have approved.
- Added RejectedApproverIds (VARCHAR(500)) to track which approvers have rejected.
- Added ApproverEmails (VARCHAR(500)) to SPRApprovers table for storing multiple approver emails.
- Added MinAmount and MaxAmount (MONEY) to SPRApprovers table for custom price range definitions.

```
[StringLength(500)]
public string? AssignedApproverIds { get; set; } // All approvers assigned to this request

[StringLength(500)]
public string? ApprovedApproverIds { get; set; } // Approvers who have approved

[StringLength(500)]
public string? RejectedApproverIds { get; set; } // Approvers who have rejected
```



Why This Was Needed:

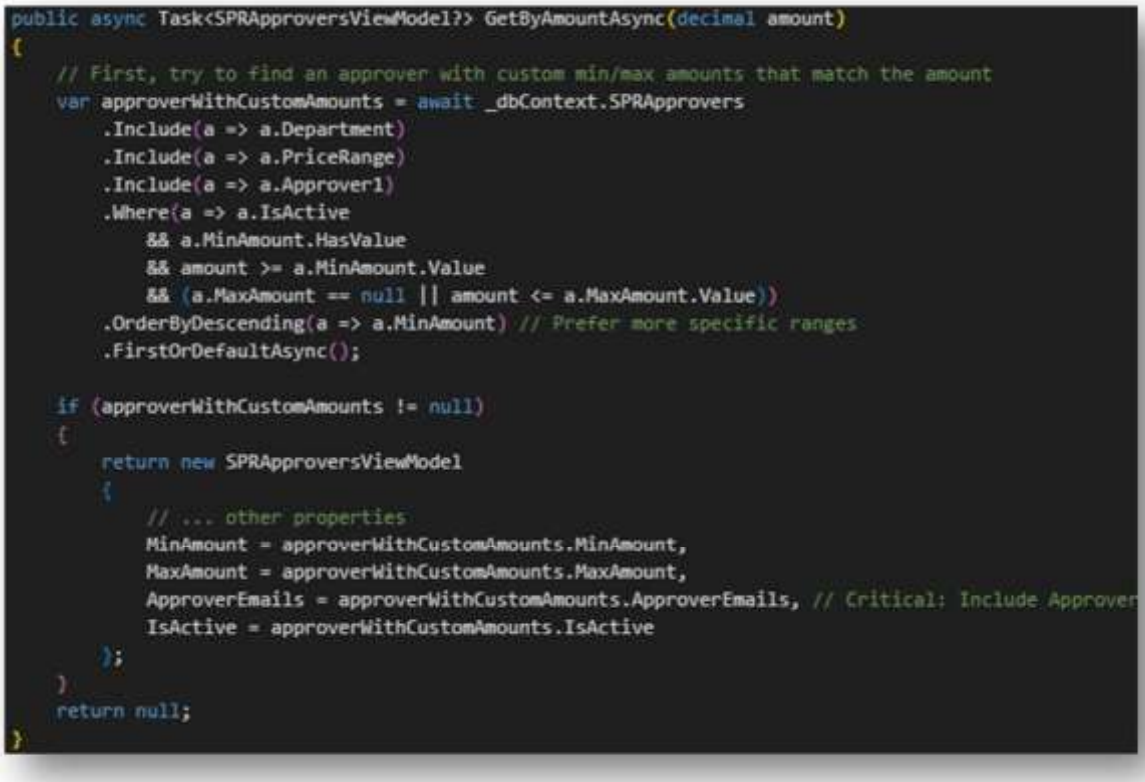
The system needed to support multiple approvers for a single SPR request, where all approvers must approve before moving to the next stage. The previous single-approver model couldn't track individual approval statuses or handle scenarios where multiple approvers are assigned to the same price range.

TASK 2: Custom Price Range Configuration for SPR Approvers

Implemented custom min/max amount fields in SPR Approvers configuration, allowing administrators to define flexible price ranges per approver configuration instead of relying solely on predefined PriceRange categories.
Implementation:

The SPRApprovers table now supports both traditional PriceRangeId-based assignment and custom MinAmount/MaxAmount ranges. When a vetter approves at Stage 6 (Vetter Credit Review), the system first attempts to find an approver using custom amounts, then falls back to PriceRange lookup.

Min Amount	Max Amount	Approvers
-	No limit	admin@delegateit.co.za
R10 000,00	R100 000,00	admin@delegateit.co.za; user3@delegateit.co.za



```
public async Task<SPRApproversViewModel?> GetByAmountAsync(decimal amount)
{
    // First, try to find an approver with custom min/max amounts that match the amount
    var approverWithCustomAmounts = await _dbContext.SPRApprovers
        .Include(a => a.Department)
        .Include(a => a.PriceRange)
        .Include(a => a.Approver1)
        .Where(a => a.IsActive
            && a.MinAmount.HasValue
            && amount >= a.MinAmount.Value
            && (a.MaxAmount == null || amount <= a.MaxAmount.Value))
        .OrderByDescending(a => a.MinAmount) // Prefer more specific ranges
        .FirstOrDefaultAsync();

    if (approverWithCustomAmounts != null)
    {
        return new SPRApproversViewModel
        {
            // ... other properties
            MinAmount = approverWithCustomAmounts.MinAmount,
            MaxAmount = approverWithCustomAmounts.MaxAmount,
            ApproverEmails = approverWithCustomAmounts.ApproverEmails, // Critical: Include Approver
            IsActive = approverWithCustomAmounts.IsActive
        };
    }
    return null;
}
```

Function Explanation:

The GetByAmountAsync method searches for active approver configurations where the request's total amount falls within the configured MinAmount and MaxAmount range. It orders results by MinAmount descending to prioritize more specific ranges. Previously, this method didn't return ApproverEmails, causing multi-approver detection to fail. The fix ensures ApproverEmails is included in the ViewModel, enabling the system to detect and assign multiple approvers.

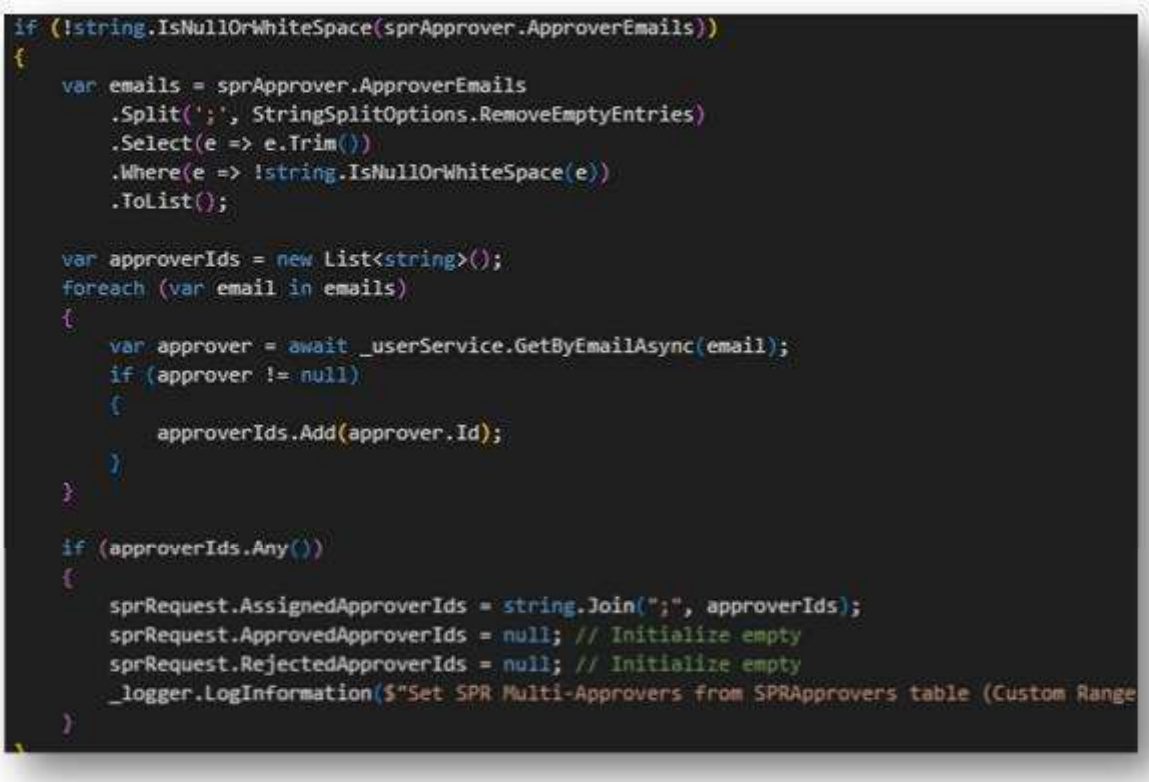
Why This Was Needed:

Administrators needed flexibility to define custom price ranges (e.g., R10,000 - R100,000) without being constrained by predefined PriceRange categories. This allows for more granular control over which approvers handle specific price ranges.

TASK 3: Multi-Approver Assignment at Vetter Credit Review Stage

Updated VetterCreditReviewApproveAsync to assign multiple approvers when a request moves to Stage 7 (Approver Review), based on the approver configuration matching the request's total amount.
Implementation:

When a vetter approves at Stage 6, the system looks up the appropriate approver configuration using either custom amounts or PriceRange. If the configuration has multiple approvers (stored in ApproverEmails), all approver IDs are extracted and stored in AssignedApproverIds.



```
if (!string.IsNullOrWhiteSpace(sprApprover.ApproverEmails))
{
    var emails = sprApprover.ApproverEmails
        .Split(';', StringSplitOptions.RemoveEmptyEntries)
        .Select(e => e.Trim())
        .Where(e => !string.IsNullOrWhiteSpace(e))
        .ToList();

    var approverIds = new List<string>();
    foreach (var email in emails)
    {
        var approver = await _userService.GetByEmailAsync(email);
        if (approver != null)
        {
            approverIds.Add(approver.Id);
        }
    }

    if (approverIds.Any())
    {
        sprRequest.AssignedApproverIds = string.Join(";", approverIds);
        sprRequest.ApprovedApproverIds = null; // Initialize empty
        sprRequest.RejectedApproverIds = null; // Initialize empty
        _logger.LogInformation($"Set SPR Multi-Approvers from SPRApprovers table (Custom Range
    }
}
```

Function Explanation:

When multiple approvers are configured (semicolon-separated emails in ApproverEmails), the system splits the string, looks up each approver by email, collects their user IDs, and stores them as a semicolon-separated string in AssignedApproverIds. This enables tracking which approvers are assigned to the request and their individual approval statuses.

Why This Was Needed:

The system needed to support scenarios where multiple approvers must review and approve a request based on its price range. This ensures all assigned approvers are notified and can track the request's progress.

TASK 4: Multi-Approver Approval Workflow Logic

Modified Approver1ApproveAsync to handle multi-approver scenarios where all approvers must approve before the request moves to Stage 9 (Credit Note Upload). If only some approvers have approved, the request remains in Stage 7 until all have approved.
Implementation:

The approval logic checks if multiple approvers are assigned. If so, it tracks individual approvals in ApprovedApproverIds. Only when all assigned approvers have approved does the request move to Stage 9. Otherwise, it stays in Stage 7 and NextApproverId is set to the next pending approver.

```
if (hasMultipleApprovers)
{
    // Multi-approver workflow - track individual approvals
    var assignedIds = sprRequest.AssignedApproverIds!
        .Split(';', StringSplitOptions.RemoveEmptyEntries)
        .Select(id => id.Trim())
        .Where(id => !string.IsNullOrWhiteSpace(id))
        .ToList();

    var approvedIds = string.IsNullOrWhiteSpace(sprRequest.ApprovedApproverIds)
        ? new List<string>()
        : sprRequest.ApprovedApproverIds!
            .Split(';', StringSplitOptions.RemoveEmptyEntries)
            .Select(id => id.Trim())
            .Where(id => !string.IsNullOrWhiteSpace(id))
            .ToList();

    // Add current approver to approved list if not already there
    if (!approvedIds.Contains(currentUserId))
    {
        approvedIds.Add(currentUserId);
        sprRequest.ApprovedApproverIds = string.Join(";", approvedIds);
    }

    // Check if all approvers have approved
    var allApproved = assignedIds.All(id => approvedIds.Contains(id));

    if (allApproved)
    {
        // All approvers have approved - move to Stage 9: Credit Note Upload
        sprRequest.Approver1Processed = true;
        sprRequest.Approved = true;
        sprRequest.StateId = 3; // Approved
        sprRequest.StageId = 9; // Credit Note Upload
        // ... log entry and notification
    }
    else
    {
        // Not all approvers have approved yet - stay in Stage 7
        sprRequest.Approver1Processed = false; // Keep as false until all approve
        sprRequest.Approved = false; // Not fully approved yet
        sprRequest.StateId = 2; // Under Review
        sprRequest.StageId = 7; // Approver Review - stay here

        // Set next approver to the first pending one
        var pendingApprovers = assignedIds.Where(id => !approvedIds.Contains(id)).ToList();
        if (pendingApprovers.Any())
        {
            sprRequest.NextApproverId = pendingApprovers.First();
```

Function Explanation:

The Approver1ApproveAsync method handles both single and multi-approver scenarios. For multi-approver requests, it:

1. Parses AssignedApproverIds and ApprovedApproverIds into lists.
2. Adds the current approver's ID to ApprovedApproverIds if not already present.
3. Checks if all assigned approvers have approved using All() LINQ method.
4. If all approved: moves to Stage 9, sets Approved = true, and logs "Approved by All Approvers".
5. If not all approved: stays in Stage 7, sets NextApproverId to the first pending approver, and logs "Approved by Approver (Partial)" with pending count.

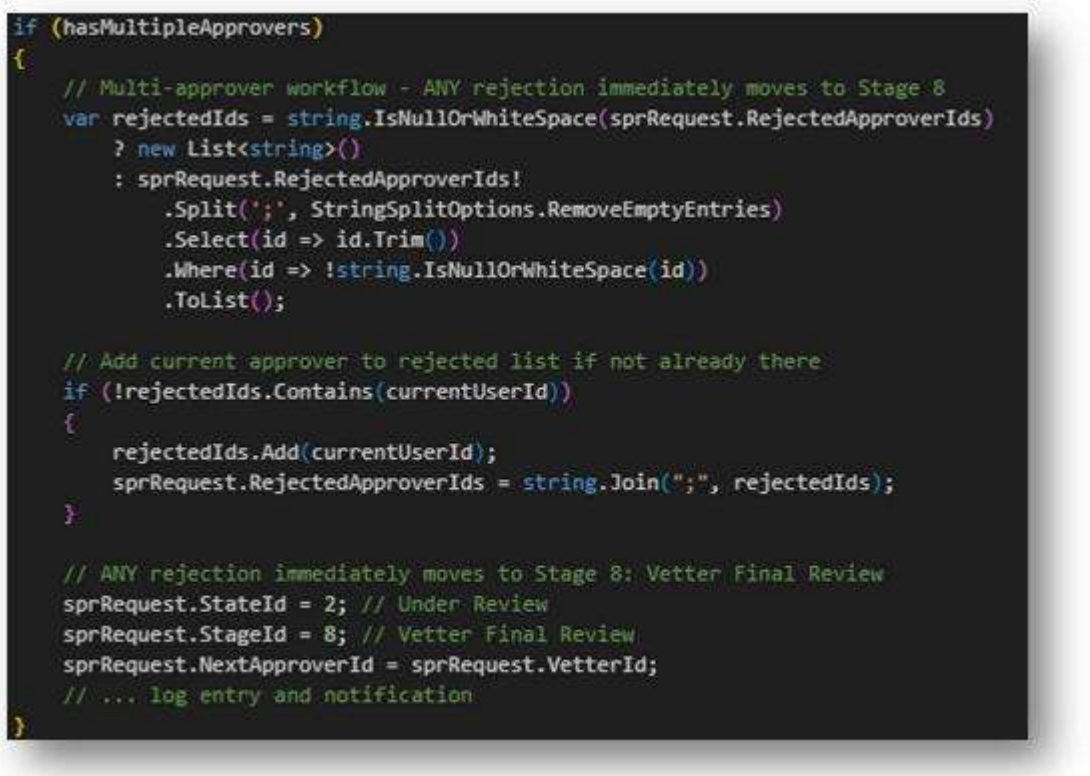
Why This Was Needed:

The workflow requires all assigned approvers to approve before proceeding. This ensures proper oversight and prevents a single approver from bypassing the multi-approver requirement.

TASK 5: Multi-Approver Rejection Workflow

Modified ApproverRejectAsync to immediately move the request to Stage 8 (Vetter Final Review) if any approver rejects, regardless of other approvers' status.
Implementation:

When any approver rejects in a multi-approver scenario, the request immediately moves to Stage 8, bypassing the need for other approvers to review. The rejecting approver's ID is added to RejectedApproverIds.



```
if (hasMultipleApprovers)
{
    // Multi-approver workflow - ANY rejection immediately moves to Stage 8
    var rejectedIds = string.IsNullOrWhiteSpace(sprRequest.RejectedApproverIds)
        ? new List<string>()
        : sprRequest.RejectedApproverIds!
            .Split(';', StringSplitOptions.RemoveEmptyEntries)
            .Select(id => id.Trim())
            .Where(id => !string.IsNullOrWhiteSpace(id))
            .ToList();

    // Add current approver to rejected list if not already there
    if (!rejectedIds.Contains(currentUserId))
    {
        rejectedIds.Add(currentUserId);
        sprRequest.RejectedApproverIds = string.Join(";", rejectedIds);
    }

    // ANY rejection immediately moves to Stage 8: Vetter Final Review
    sprRequest.StateId = 2; // Under Review
    sprRequest.StageId = 8; // Vetter Final Review
    sprRequest.NextApproverId = sprRequest.VetterId;
    // ... log entry and notification
}
```

Function Explanation:

The rejection logic for multi-approver scenarios ensures that a single rejection immediately sends the request back to the vetter for final review. This prevents wasted time if one approver identifies issues that need to be addressed before other approvers review.

Why This Was Needed:

If any approver identifies problems, the request should be sent back immediately rather than waiting for all approvers to review. This speeds up the rejection process and reduces unnecessary work.

TASK 6: Dynamic Approve Button Text Based on Approval Status

Updated the approve button text in Stage 7 to dynamically display "Approve as Approver 1" or "Approve as Approver 2" based on how many approvers have already approved, but only when multiple approvers are assigned.
Implementation:

The GetApproveButtonText() method in ViewSPRRequest.razor checks if multiple approvers are assigned and counts how many have already approved. The button text changes accordingly to provide clear context to each approver.

```
private string GetApproveButtonText()
{
    if (currentSPRRequest == null) return "Approve Request";

    // Stage 7: Check for multiple approvers
    if (currentSPRRequest.StageId == 7)
    {
        // Check if multiple approvers are assigned
        var hasMultipleApprovers = !string.IsNullOrWhiteSpace(currentSPRRequest.AssignedApproverIds);

        if (hasMultipleApprovers)
        {
            // Count how many have already approved
            var approvedCount = 0;
            if (!string.IsNullOrWhiteSpace(currentSPRRequest.ApprovedApproverIds))
            {
                approvedCount = currentSPRRequest.ApprovedApproverIds
                    .Split(';', StringSplitOptions.RemoveEmptyEntries)
                    .Count(id => !string.IsNullOrWhiteSpace(id.Trim()));
            }

            // Return appropriate text based on approval count
            return approvedCount switch
            {
                0 => "Approve as Approver 1",
                1 => "Approve as Approver 2",
                _ => "Approve & Move to Credit Note Upload" // Fallback
            };
        }
        else
        {
            // Single approver - original text
            return "Approve & Move to Credit Note Upload";
        }
    }

    // ... other stage button texts
}
```

Function Explanation:

The GetApproveButtonText() method determines the appropriate button text by:

- Checking if the request is in Stage 7 (Approver Review).
- Verifying if multiple approvers are assigned by checking if AssignedApproverIds is not empty.
- Counting approved approvers by splitting ApprovedApproverIds and counting non-empty entries.
- Returning "Approve as Approver 1" if 0 approved, "Approve as Approver 2" if 1 approved, or the default text for single approver scenarios.

Why This Was Needed:

Approvers need clear indication of their position in the approval sequence. This helps them understand whether they're the first or second approver and provides context about the approval status.

TASK 7: Multi-Approver Status Display in UI

Added a "Multi-Approver Status" section in Stage 7 that displays all assigned approvers with their individual approval/rejection status using visual indicators (checkmark for approved, cross for rejected, clock for pending).
Implementation:

The UI section loads approver details from AssignedApproverIds, checks their status against ApprovedApproverIds and RejectedApproverIds, and displays each approver with their email and status icon.

```
                var approver = approverDetails.FirstOrDefault(a => a.Id == approverId);
                var isApproved = approvedApproverIds.Contains(approverId);
                var isRejected = rejectedApproverIds.Contains(approverId);

                <div class="list-group-item">
                    <div class="d-flex justify-content-between align-items-center">
                        <span>@(approver?.Email ?? approverId)</span>
                        <span>
                            @if (isApproved)
                            {
                                <i class="fas fa-check-circle text-success" title="Approved"></i>
                            }
                            else if (isRejected)
                            {
                                <i class="fas fa-times-circle text-danger" title="Rejected"></i>
                            }
                            else
                            {
                                <i class="fas fa-clock text-warning" title="Pending"></i>
                            }
                        </span>
                    </div>
                </div>
            }
        </div>
    </div>
}
```

Function Explanation:

The multi-approver status section provides real-time visibility into the approval process. It:

- Parses AssignedApproverIds to get the list of all approvers.
- Checks each approver's ID against ApprovedApproverIds and RejectedApproverIds to determine status.
- Displays visual indicators (green checkmark for approved, red cross for rejected, orange clock for pending).
- Shows summary counts for total, approved, rejected, and pending approvers.

Why This Was Needed:

Approvers need to see the status of other approvers to understand the overall approval progress and know when their approval is needed or when the request has been rejected by another approver.

TASK 8: Todo List and Watch List Updates for Multi-Approvers

Updated GetMyToDoAsync and GetMyWatchListAsync methods to include SPR requests where the user is in the AssignedApproverIds list, ensuring all assigned approvers can see and track requests assigned to them.
Implementation:

Both methods now check if the user's ID is contained in the AssignedApproverIds field. For the todo list, it also verifies the user hasn't already approved. For the watch list, it includes all assigned approvers regardless of approval status.

```
        if (r.StageId == 7 && !string.IsNullOrEmpty(r.AssignedApproverIds))
        {
            var assignedIds = r.AssignedApproverIds
                .Split(';', StringSplitOptions.RemoveEmptyEntries)
                .Select(id => id.Trim())
                .ToList();

            if (assignedIds.Contains(userId))
            {
                // Check if user has already approved
                if (!string.IsNullOrEmpty(r.ApprovedApproverIds))
                {
                    var approvedIds = r.ApprovedApproverIds
                        .Split(';', StringSplitOptions.RemoveEmptyEntries)
                        .Select(id => id.Trim())
                        .ToList();

                    // Only show if user hasn't approved yet
                    return !approvedIds.Contains(userId);
                }
                return true; // User is assigned and hasn't approved yet
            }
        }
        return true; // Keep all other requests
    }).ToList();

    return filteredRequests.Select(r => ConvertToViewModel(r)).ToList();
}
```

Function Explanation:

The GetMyToDoAsync method:

- Initially filters requests where the user might be involved (including Stage 7 with AssignedApproverIds containing the user).
- Performs in-memory filtering to verify the user is actually in the assigned list (exact match, not substring).
- For Stage 7 multi-approver requests, only includes requests where the user hasn't already approved.
- Returns the filtered list converted to ViewModels.

```
public async Task<List<SPRRequestViewModel>> GetMyWatchListAsync(string userId)
{
    var requests = await _dbContext.SPRRequests
        .Include(r => r.Dealer)
        .Include(r => r.Branch)
        .Include(r => r.Requestor)
        .Include(r => r.Stage)
        .Include(r => r.State)
        .Include(r => r.SPRPartItems)
        .Where(r => r.VetterId == userId ||
                    r.Approver1Id == userId ||
                    r.RequestorId == userId ||
                    (!string.IsNullOrEmpty(r.AssignedApproverIds) && r.AssignedApproverIds.Contai
        .ToListAsync();

    // Additional filtering for multi-approver: ensure user is actually in the list
    var filteredRequests = requests.Where(r =>
    {
        if (!string.IsNullOrEmpty(r.AssignedApproverIds))
        {
            var assignedIds = r.AssignedApproverIds
                .Split(';', StringSplitOptions.RemoveEmptyEntries)
                .Select(id => id.Trim())
                .ToList();

            if (assignedIds.Contains(userId))
            {
                return true; // User is assigned, include in watchlist
            }
        }

        // Keep all other requests (vetter, approver1, requestor)
        return r.VetterId == userId || r.Approver1Id == userId || r.RequestorId == userId;
    }).ToList();

    return filteredRequests.Select(r => ConvertToViewModel(r)).ToList();
```

Function Explanation:

The GetMyWatchListAsync method:

- Initially filters requests where the user might be involved (including any request with AssignedApproverIds containing the user).
- Performs in-memory filtering to verify exact match in the assigned list.
- Includes all assigned approvers in the watch list regardless of approval status (so they can track progress)..
- Also includes traditional vetter, approver1, and requestor assignments

Why This Was Needed:

All assigned approvers need visibility into requests assigned to them. The todo list shows pending actions, while the watch list allows tracking progress even after approval. This ensures approvers can monitor requests throughout the approval process.

Summary

The multi-approver workflow enables:

- Multiple approvers assigned to a single SPR request based on price range.
- All approvers must approve before moving to Stage 9.
- Any approver rejection immediately sends the request to Stage 8.
- Dynamic button text indicating approval position.
- Real-time status display showing all approvers' status.
- Todo and watch list visibility for all assigned approvers.
- Custom price range configuration with min/max amounts.
- Flexible approver assignment based on request amount.

PROJECT 2: MAN Trucks & Bus TCO Tool

TASK 1: Fuel and AdBlue Calculation Reactivity Fix for MAN 1

Fixed a critical bug where changing inputs for "Total Fuel Consumption" and "Fuel Price" in the MAN 1 section did not dynamically update the "Fuel Cost + AdBlue" display. The system was using fallback values instead of prioritizing the global financial parameters that users were actively editing.
Implementation:

The issue was traced to the calculation logic that retrieved fuel consumption and fuel price values. For MAN 1 (index 0), the system was checking manVehicleFuelConsumptions[index] first, which could be undefined or stale, before falling back to financialParams. This was fixed by implementing conditional logic that ensures MAN 1 always uses financialParams directly since those are the inputs being edited by the user.

```
const fuelConsumption = index === 0
  ? financialParams.fuelConsumption
  : (manVehicleFuelConsumptions[index] || financialParams.fuelConsumption);

const fuelPrice = index === 0
  ? financialParams.fuelPrice
  : (manVehicleFuelPrices[index] || financialParams.fuelPrice);

// Calculate fuel cost
const kmPerLt = fuelConsumption > 0 ? (100 / fuelConsumption) : 0;
const totalFuelConsumption = kmPerLt > 0 ? (kmTotal / kmPerLt) : 0;
const fuelCost = totalFuelConsumption * fuelPrice;

// Calculate AdBlue cost
const adBlueUsage = index === 0
  ? (financialParams.adBlueUsage || 0)
  : (manVehicleAdBlueUsages[index] || financialParams.adBlueUsage || 0);
const adBluePrice = index === 0
  ? (financialParams.adBluePrice || 0)
  : (manVehicleAdBluePrices[index] || financialParams.adBluePrice || 0);
const adBlueCost = adBlueUsage * adBluePrice;

return fuelCost + adBlueCost;
```

Function Explanation:

The conditional logic checks if the current vehicle index is 0 (MAN 1). If so, it directly uses financialParams.fuelConsumption and financialParams.fuelPrice which are bound to the input fields the user is editing. For additional vehicles (index > 0), it checks vehicle-specific state arrays first before falling back to financial parameters. This ensures that when users modify the fuel consumption or fuel price inputs, React's state update triggers a re-render with the new values immediately reflected in the Fuel Cost + AdBlue calculation.

Why This Was Needed:

Users expected real-time calculation updates when modifying input fields. The original implementation's fallback chain meant MAN 1 could use stale or undefined values from vehiclespecific arrays instead of the actively-edited global parameters, breaking the reactive user experience.

TASK 2: Total Fuel Consumption Display Correction

Fixed an issue where the "Total Fuel Consumption" for MAN 1 was displaying incorrect values (800,000 L instead of the expected 300,000 L). The calculation was using incorrect kilometer totals from warranty lookups rather than the workflow data.
Implementation:

The kmTotal retrieval logic was updated to prioritize workflow data for MAN 1. The system now checks manVehicleKmTotals[index] or connectedDealData?.workflowData?.tcoQuotation?.ter mKms for MAN 1 before performing any warranty-based lookups.

```
let kmTotal = index === 0 && (manVehicleKmTotals[index] || connectedDealData?.workflowData?.
  ? (manVehicleKmTotals[index] || connectedDealData?.workflowData?.tcoQuotation?.termKms ||
  : (manVehicleKmTotals[index] || 0);

// Use the correct fuel consumption based on vehicle index
const fuelConsumption = index === 0
  ? financialParams.fuelConsumption
  : (manVehicleFuelConsumptions[index] || financialParams.fuelConsumption);

const kmPerLt = fuelConsumption > 0 ? (100 / fuelConsumption) : 0;
return kmPerLt > 0 ? Math.round(kmTotal / kmPerLt).toLocaleString() + ' L' : '0 L';
```

Function Explanation:

The function first determines the correct kmTotal by checking if this is MAN 1 (index 0) and if workflow data exists. For MAN 1, it prioritizes the TCO quotation term kilometers from the connected deal data, which represents the actual contract kilometers entered in the workflow. The fuel consumption value is then sourced correctly based on whether this is the primary vehicle or an additional vehicle. The calculation converts kilometers to liters using the L/100km fuel consumption rate and formats the result with locale-appropriate thousand separators.

Why This Was Needed:

The original implementation was performing redundant warranty lookups that returned incorrect kilometer values, causing the fuel consumption calculation to use 800,000 km instead of the workflow's 300,000 km term kilometers.

TASK 3: Competitor Fuel + AdBlue CPK Calculation Fix

Resolved a persistent issue where "Fuel + AdBlue CPK" for competitor vehicles displayed R0 instead of calculated values. The competitor calculations were using static object properties rather than reactive state variables.

Implementation:

The calculation was refactored to use state variables:

- competitorVehicleFuelConsumptions[index]
- competitorVehicleFuelPrices[index]
- competitorVehicleKmTotals[index])

Instead of directly accessing object properties like competitor.fuelConsumptionL100km or competitor.kmTotal, and a fallback calculation was added for cases where kmTotal is zero.

```
let kmTotal = competitorVehicleKmTotals[index] || competitor.kmTotal || 0;
if (kmTotal === 0) {
  const termMonths = competitorVehicleTermMonths[index] || financialParams.period || 60;
  const kmPerMonth = competitorVehicleKmPerMonths[index] || financialParams.kmPerMonth || 5000;
  kmTotal = termMonths * kmPerMonth;
}

// Use state variables for fuel parameters
const fuelConsumption = competitorVehicleFuelConsumptions[index] || competitor.fuelConsumptionL100km
const fuelPrice = competitorVehicleFuelPrices[index] || competitor.fuelPricePerLitre || 20.00;
const adBlueUsage = competitorVehicleAdBlueUsages[index] || competitor.adBlueUseLt || 0;
const adBluePrice = competitorVehicleAdBlueCosts[index] || competitor.adBlueCost || 0;

// Calculate CPK
const kmPerLitre = 100 / fuelConsumption;
const totalFuelConsumption = kmTotal / kmPerLitre;
const fuelCost = totalFuelConsumption * fuelPrice;
const adBlueCost = adBlueUsage * adBluePrice;
const totalCost = fuelCost + adBlueCost;
const fuelAdBlueCPK = kmTotal > 0 ? totalCost / kmTotal : 0;
```

Function Explanation:

The function first retrieves the kilometer total from the competitor state array, falling back to the competitor object property, then to zero. If zero, it calculates kmTotal from term months and kilometers per month to ensure a valid denominator. Each fuel parameter (consumption, price, AdBlue usage, AdBlue price) is sourced from state arrays first, then falls back to competitor object properties, then to sensible defaults. The CPK calculation divides total fuel and AdBlue cost by kilometers, returning zero if kmTotal is zero to prevent division errors.

Why This Was Needed:

Competitor input fields were updating state variables, but the calculation was reading from the original competitor object which remained unchanged. This disconnect caused the display to show R0 regardless of user input.

TASK 4: Competitor Total Cost Per Kilometer Enhancement

Enhanced the competitor "TOTAL COST PER KILOMETER" display to correctly include both Finance CPK and Fuel + AdBlue CPK components. Previously, the total was only reflecting the finance component (R1.24), omitting fuel costs entirely.
Implementation:

The display was updated to calculate and sum both the finance CPK (monthly payment divided by monthly kilometers) and the fuel + AdBlue CPK, presenting the true total cost per kilometer for competitor vehicles.

```
const financeCPK = (() => {
  const monthlyPayment = calculatePMT(
    competitorVehicleInterestRates[index] || 12.25,
    competitorVehicleTermMonths[index] || 60,
    totalFinanced,
    0
  );
  const kmPerMonth = competitorVehicleKmPerMonths[index] || financialParams.kmPerMonth || 5000;
  return kmPerMonth > 0 ? monthlyPayment / kmPerMonth : 0;
})();

// Calculate Fuel + AdBlue CPK (from previous calculation)
const fuelAdBlueCPK = /* calculation from Task 3 */;

// Total CPK is the sum of both components
const totalCPK = financeCPK + fuelAdBlueCPK;

// Display
<td className="px-4 py-2 text-right">
  R{totalCPK.toFixed(2)} Per Kilometer Total Cost
</td>
```

Function Explanation:

The finance CPK calculation uses the PMT (Payment) function to determine monthly payments based on interest rate, term, and financed amount, then divides by kilometers per month. The fuel + AdBlue CPK is calculated separately as shown in Task 3. The total CPK sums both components to give users an accurate picture of the complete cost per kilometer for competitor vehicles, enabling meaningful comparison with MAN vehicles.

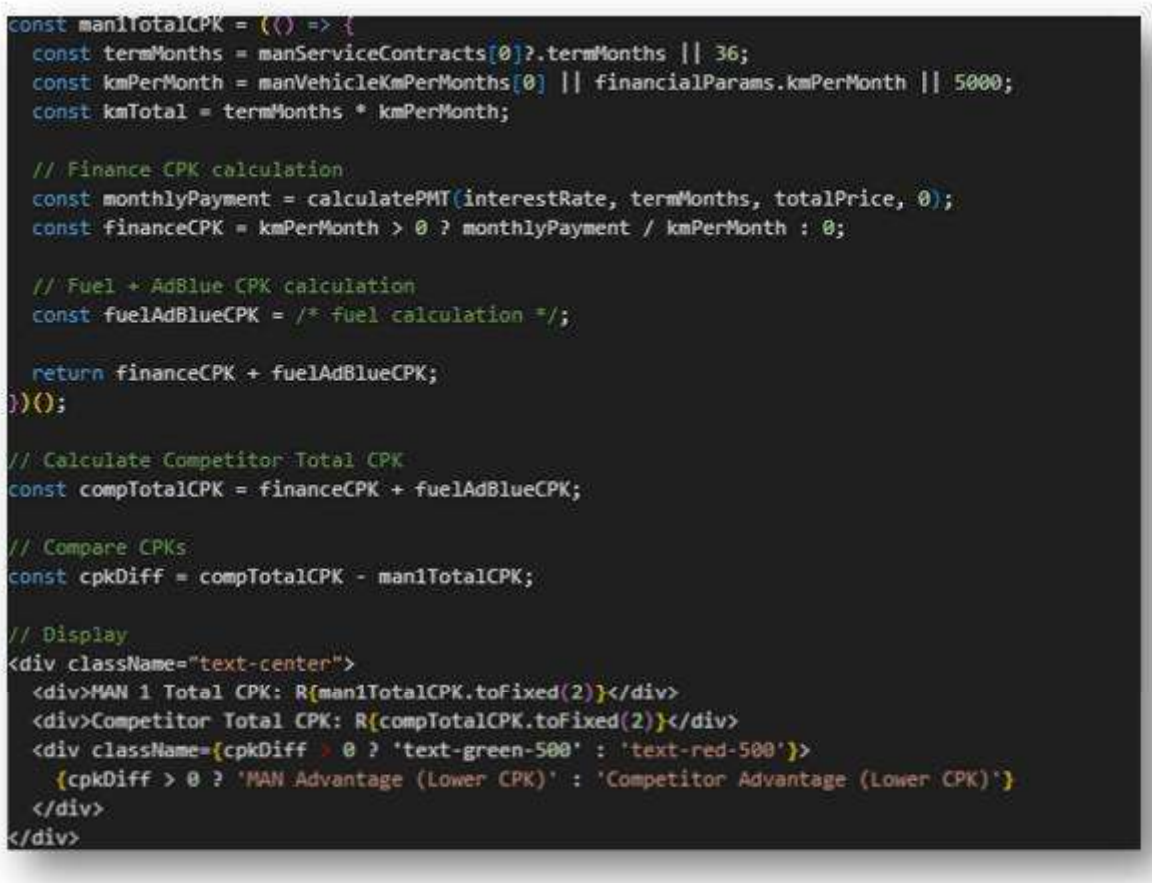
Why This Was Needed:

The original display only showed finance CPK, which significantly understated the true operating cost of competitor vehicles. Users needed the complete cost picture for accurate TCO comparisons.

TASK 5: Competitive Analysis Section Overhaul

Modified the Competitive Analysis comparison to evaluate vehicles based on Total Cost Per Kilometer rather than absolute ZAR amounts, and updated the display to show "MAN Advantage" or "Competitor Advantage" without exposing raw currency figures.
Implementation:

The comparison logic was changed from calculating vsManCost (total cost di erence in ZAR) to calculating cpkDi (di erence in Cost Per Kilometer). The display was updated to show both CPK values and indicate which vehicle has the advantage based on lower CPK.



```
const man1TotalCPK = (() => {
  const termMonths = manServiceContracts[0]?.termMonths || 36;
  const kmPerMonth = manVehicleKmPerMonths[0] || financialParams.kmPerMonth || 5000;
  const kmTotal = termMonths * kmPerMonth;

  // Finance CPK calculation
  const monthlyPayment = calculatePMT(interestRate, termMonths, totalPrice, 0);
  const financeCPK = kmPerMonth > 0 ? monthlyPayment / kmPerMonth : 0;

  // Fuel + AdBlue CPK calculation
  const fuelAdBlueCPK = /* fuel calculation */;

  return financeCPK + fuelAdBlueCPK;
})();

// Calculate Competitor Total CPK
const compTotalCPK = financeCPK + fuelAdBlueCPK;

// Compare CPKs
const cpkDiff = compTotalCPK - man1TotalCPK;

// Display
<div className="text-center">
  <div>MAN 1 Total CPK: R{man1TotalCPK.toFixed(2)}</div>
  <div>Competitor Total CPK: R{compTotalCPK.toFixed(2)}</div>
  <div className={cpkDiff > 0 ? 'text-green-500' : 'text-red-500'}>
    {cpkDiff > 0 ? 'MAN Advantage (Lower CPK)' : 'Competitor Advantage (Lower CPK)'}
  </div>
</div>
```

Function Explanation:

The function calculates the total CPK for both MAN 1 and the competitor vehicle using identical logic (finance CPK + fuel/AdBlue CPK). It then computes the difference, where a positive value indicates MAN has the lower (better) CPK. The display shows both CPK values for transparency and color-codes the advantage indicator green for MAN advantage or red for competitor advantage.

Why This Was Needed:

Comparing total costs in ZAR wasn't meaningful without context. CPK provides a normalized metric that accounts for different contract lengths and usage patterns, enabling fair comparison regardless of deal structure.

PROJECT 1: MAN TCO

TASK 1: Extras tick box to control quote display (non-ticked extras will not appear on quote)

Implemented a toggle control system for extras visibility in generated quote documents, allowing users to selectively include or exclude individual extras from quote documentation while maintaining complete extras data in the deal calculation. The implementation required adding a showInQuote boolean flag to the extras data structure and modifying quote generation logic to filter extras based on this flag before sending data to the document template.

Previously, all extras added to a deal were automatically included in generated quote documents, with no mechanism to exclude specific extras from customer-facing documentation while retaining them in internal calculations. This limitation caused issues when certain extras needed to be tracked internally for costing purposes but should not be disclosed to customers in the quote documentation. Sales personnel frequently needed to manually edit generated quote documents to remove specific extras, creating additional work and increasing the risk of documentation inconsistencies.

The extras management interface was enhanced with a checkbox column in the extras summary table that displays a tick box alongside each extra item. The checkbox is checked by default for all extras, ensuring backward compatibility with existing deals where all extras should appear in quotes. Users can uncheck the box for any extra to exclude it from quote generation while keeping the extra in the deal for internal calculation purposes. The checkbox state is bound to the showInQuote property of each extra item, which is stored as part of the extras data structure in the database.

The quote generation filtering logic was implemented throughout the quote data preparation functions to filter the extrasData array based on the showInQuote flag before calculating totals or sending data to the template. This filtering occurs at multiple points in the quote generation workflow, including the ExtrasTotal calculation, SubtotalWithExtras calculation, FinalTotal calculation, and VAT calculations, ensuring that excluded extras do not affect any customer-facing pricing calculations in the quote document.

The implementation maintains a clear separation between internal deal calculations (which always include all extras regardless of showInQuote status) and customer-facing quote documentation (which only includes extras where showInQuote is true). This allows sales personnel to maintain accurate internal costing while presenting simplified quotes to customers that exclude specific extras such as internal administrative fees, dealer preparation costs, or other items that should not be disclosed in customer documentation.

The filtering code uses JavaScript array filter methods to exclude extras where showInQuote is explicitly set to false, with a default behavior that includes extras where showInQuote is undefined or null to maintain backward compatibility with existing extras data that was created before this feature was implemented. The code pattern `(data.extrasData || []).filter((extra: any) => extra.showInQuote !== false)` appears throughout the quote generation logic, ensuring consistent filtering behavior across all quote calculations.

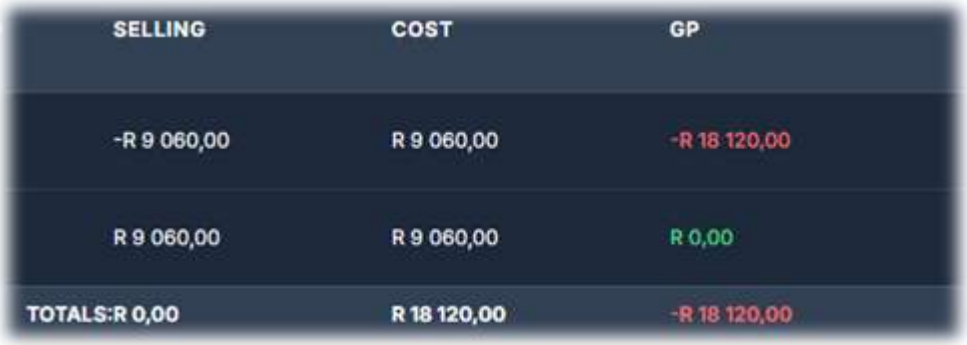
```
const updateExtra = (extraId: string, field: string, value: any) => {
  const updatedExtras = dealExtras.map((extra) => {
    if (extra.id === extraId) {
      const updatedExtra = { ...extra, [field]: value };

      // Auto-calculate GP when selling or cost changes
      if (field === 'sellingPrice' || field === 'cost') {
        const selling = field === 'sellingPrice' ? parseFloat(value) || 0 : extra.sellingPrice;
        const cost = field === 'cost' ? parseFloat(value) || 0 : extra.cost;
        updatedExtra.grossProfit = selling - cost;
      }

      return updatedExtra;
    }
    return extra;
  });
  setDealExtras(updatedExtras);
};
```

TASK 2: Total GP color coding (red when negative)

Implemented color-coded visual indicators for gross profit (GP) calculations throughout the TCO Calculator interface, automatically displaying negative GP values in red to provide immediate visual feedback when deals are operating at a loss. The implementation required identifying all GP calculation display points and applying conditional styling based on whether the GP value is positive or negative.



SELLING	COST	GP
-R 9 060,00	R 9 060,00	-R 18 120,00
R 9 060,00	R 9 060,00	R 0,00
TOTALS:R 0,00	R 18 120,00	-R 18 120,00

Previously, gross profit values were displayed in standard white or gray text regardless of whether they were positive or negative, making it difficult for sales personnel and managers to quickly identify deals that were losing money or operating below acceptable profit margins. This lack of visual differentiation required users to carefully read each GP value to determine profitability status, slowing down deal review processes and increasing the risk of approving deals with negative margins.

The color-coding system applies red text styling (text-red-400 or text-red-500 depending on context) to any GP value that is less than zero, while maintaining standard white or green text styling for positive GP values. The conditional styling uses JavaScript ternary operators or template literals to dynamically apply Tailwind CSS classes based on the calculated GP value: `className={GP < 0 ? "text-red-400" : "text-white"}`.

The implementation extends across multiple components including the Extras step where extras profit/loss is displayed, the Discount step where OBR breakdown shows provision calculations, and the final TCO calculator where total GP and various provision calculations are presented. Each display location checks the relevant GP or provision value and applies appropriate color coding to ensure consistent visual feedback throughout the application.

The visual feedback system significantly improves deal review efficiency by allowing managers and administrators to instantly identify problematic deals through visual scanning rather than numerical analysis. The red coloring creates an immediate visual alert that draws attention to negative margins, supporting faster decision-making and reducing the risk of approving deals that operate at a loss.

The color coding is particularly valuable in list views and summary tables where multiple deals are displayed simultaneously, enabling quick identification of deals requiring closer review or margin adjustment.

```
// Calculate total gross profit function
const getTotalGrossProfit = useCallback(() => {
  return dealExtras.reduce((total, extra) => total + extra.grossProfit, 0);
}, [dealExtras]);

{/* Desktop Table Footer - Total GP */}
<div className="bg-slate-700 border-t border-slate-600">
  <div className="grid grid-cols-9 px-4 py-2 text-xs font-bold">
    <div className="col-span-4 text-right text-white">TOTALS:</div>
    <div className="text-white">{formatCurrency(getTotalExtrasValue())}</div>
    <div className="text-white">
      {formatCurrency(dealExtras.reduce((total, extra) => total + extra.cost, 0))}
    </div>
    <div className={`font-semibold ${
      getTotalGrossProfit() >= 0 ? 'text-green-400' : 'text-red-400'
    }`}>
      {formatCurrency(getTotalGrossProfit())}
    </div>
    <div></div>
    <div></div>
  </div>
</div>

{/* Mobile Summary Card - Total GP */}
<div>
  <span className="block text-xs font-medium text-slate-300">Gross Profit</span>
  <span className={`font-semibold ${
    getTotalGrossProfit() >= 0 ? 'text-green-400' : 'text-red-400'
  }`}>
    {formatCurrency(getTotalGrossProfit())}
  </span>
</div>
```

TASK 3: Duplicate MAN Vehicle (MAN 2+) inherits MAN 1 values, independently configurable

Implemented comprehensive duplicate vehicle functionality in the TCO Calculator, enabling users to add multiple MAN vehicles (MAN 2, MAN 3, etc.) for comparative analysis, with each duplicate vehicle inheriting initial values from MAN 1 but remaining independently configurable for kilometers total, service level, contract term (months), and deployment type. The implementation required extending the TCO calculator data structure to support multiple MAN vehicles while ensuring each vehicle maintains its own independent configuration and calculation results.

Previously, the TCO Calculator only supported a single MAN vehicle for comparison against competitor vehicles, limiting the system's ability to perform internal MAN vehicle comparisons across different configurations. Sales personnel frequently needed to create multiple separate deals to compare different MAN vehicle configurations (such as different service levels or contract terms), which was time-consuming and prevented side-by-side comparison within a single TCO analysis.

The duplicate vehicle creation mechanism initializes each new MAN vehicle (MAN 2, MAN 3, etc.) with identical values from MAN 1, including chassis code, retail price, discount percentage, extras, warranty configuration, trade back status, and financial parameters. This ensures that users start with a consistent baseline configuration when creating comparative scenarios, while allowing modifications to specific parameters for comparison purposes.

The independent configuration system allows each MAN vehicle to have its own kilometers total, service level, contract term (months), and deployment type parameters that directly affect its maintenance calculations, warranty costs, fuel consumption, and total TCO. When a user changes the kilometers total for MAN 2, only MAN 2's TCO recalculates, without affecting MAN 1 or other MAN vehicles. Similarly, changing the service level from "Comfort Care" to "Extreme Care" for MAN 3 updates only MAN 3's maintenance costs, allowing precise comparative analysis of service level impact.

The dynamic calculation engine recalculates TCO for each MAN vehicle independently based on its specific configuration, enabling accurate comparison scenarios such as comparing a 36-month contract at 300,000 km (MAN 1) versus a 48-month contract at 400,000 km (MAN 2) versus a 60-month contract at 600,000 km (MAN 3), all within a single TCO analysis. Each vehicle maintains its own calculation results including total monthly payment, maintenance cost per kilometer, fuel costs, insurance, and total TCO, which are displayed side-by-side in the comparative analysis table.

The implementation maintains data integrity by ensuring that each MAN vehicle's configuration is stored separately in the manVehicles array, with each vehicle having its own index (0 for MAN 1, 1 for MAN 2, etc.) that is used consistently across all calculation functions and display components. The system supports adding, configuring, and removing duplicate MAN vehicles dynamically during deal creation or editing, providing flexibility in comparative analysis scenarios.

```
// Main MAN vehicles array - stores all MAN vehicles (MAN 1, MAN 2, MAN 3, etc.)
const [manVehicles, setManVehicles] = useState<ManVehicle[]>([]);

// Track which vehicles are duplicates (MAN 2+)
const [manVehicleIsDuplicate, setManVehicleIsDuplicate] = useState<{ [vehicleIndex: number]: boolean }>({});

// Independent configuration state for each vehicle by index
const [manVehicleKmTotals, setManVehicleKmTotals] = useState<{ [vehicleIndex: number]: number }>({
  0: 600000 // MAN 1 initialized from workflow
});

const [manVehicleServiceLevels, setManVehicleServiceLevels] = useState<{ [vehicleIndex: number]: string }>({
  0: 'Comfort Care' // MAN 1 initialized from workflow
});

const [manVehicleTermMonths, setManVehicleTermMonths] = useState<{ [vehicleIndex: number]: number }>({
  0: 36 // MAN 1 initialized from workflow
});

const [manVehicleDeploymentTypes, setManVehicleDeploymentTypes] = useState<{
  [vehicleIndex: number]: 'LONG HAUL' | 'TRACTOR TIPPER'
}>({});

// Financial parameters - independent per vehicle
const [manVehicleRetailPrices, setManVehicleRetailPrices] = useState<{ [vehicleIndex: number]: number }>({
  0: 0 // MAN 1 initialized from workflow
});

const [manVehicleDiscountPercents, setManVehicleDiscountPercents] = useState<{ [vehicleIndex: number]: number }>({
  0: 0 // MAN 1 initialized from workflow
});

// Extras and services - independent per vehicle
const [manVehicleExtras, setManVehicleExtras] = useState<{ [vehicleIndex: number]: DealExtra[] }>({
  0: [] // MAN 1 initialized from workflow
});

const [manVehicleHandoverServices, setManVehicleHandoverServices] = useState<{
  [vehicleIndex: number]: HandoverService[]
}>({
  0: [] // MAN 1 initialized from workflow
});
```

TASK 4: Competitive Analysis displays MAN 2 comparisons alongside MAN 1 and Competitors

Enhanced the Competitive Analysis component to display all MAN vehicles (MAN 1, MAN 2, MAN 3, etc.) in side-by-side comparison with competitor vehicles, enabling comprehensive multi-vehicle analysis within a single TCO report. The implementation required restructuring the comparative analysis table to support multiple MAN vehicles rather than assuming a single MAN vehicle comparison model.

Previously, the Competitive Analysis section only compared MAN 1 against competitor vehicles, effectively hiding MAN 2+ vehicles from the comparison even when they were configured in the TCO calculator. This prevented users from performing internal MAN vehicle comparisons to determine optimal configurations before comparing against competitors, limiting the analytical value of the duplicate vehicle functionality.

The comparative analysis table was redesigned to dynamically generate columns for all configured MAN vehicles, displaying MAN 1, MAN 2, MAN 3, etc. in sequential order before competitor columns. Each MAN vehicle column displays its complete TCO breakdown including vehicle price, discount, extras, maintenance costs, fuel costs, insurance, and total TCO, enabling side-by-side comparison of different MAN configurations alongside competitor offerings.

The column generation logic iterates through the manVehicles array to create a column for each configured MAN vehicle, using the vehicle index to identify and label each column appropriately (MAN 1, MAN 2, MAN 3). The system maintains visual consistency with MAN-themed red gradient headers for MAN vehicle columns while using blue gradient headers for competitor columns, providing clear visual differentiation between internal and external comparisons.

The savings calculation was extended to support multi-MAN vehicle scenarios, calculating savings not only between MAN 1 and competitors but also between different MAN configurations. This enables users to identify the optimal MAN vehicle configuration based on total cost of ownership before comparing against competitor offerings, supporting data-driven decision-making in vehicle selection and configuration optimization.

The implementation ensures that all comparative analysis features (total TCO per vehicle, TCO per kilometer, TCO per year, monthly payment breakdowns, savings calculations) work correctly with multiple MAN vehicles, maintaining calculation accuracy regardless of how many MAN vehicles are configured in the analysis. The visual comparison supports up to 10 MAN vehicles simultaneously, though the recommended maximum is 3-4 vehicles to maintain readable table layouts.

```jsx
{/* Table Header - Dynamically generates columns for all MAN vehicles + Competitors */}
<thead>
  <tr className="bg-gray-100">
    {/* Specification Column */}
    <th className="border-r border-gray-900 px-3 py-2 text-xs font-bold text-left min-w-[150px]">
      SPECIFICATION
    </th>

    {/* ✅ MAN Vehicle Columns - Generated dynamically from manVehicles array */}
    {manVehicles.map((manVehicle, index) => (
      <th
        key={`man-${index}`}
        className="border-r border-gray-900 px-3 py-2 text-xs font-bold text-center bg-man-red-50 min-w-[120px]"
      >
        <div className="text-man-red-600 font-bold">MAN {index + 1}</div>
        <div className="text-xs font-normal text-man-red-500 mt-1">
          {manVehicle.chassisCode || 'N/A'}
        </div>
      </th>
    ))}

    {/* Competitor Columns - Generated dynamically from competitorData array */}
    {competitorData.map((competitor, index) => (
      <th
        key={`comp-${index}`}
        className="border-r border-gray-900 px-3 py-2 text-xs font-bold text-center bg-blue-50 min-w-[120px]"
        style={{ borderRight: index === competitorData.length - 1 ? 'none' : '1px solid #111827' }}
      >
        <div className="text-blue-600 font-bold">Competitor {index + 1}</div>
        <div className="text-xs font-normal text-blue-500 mt-1">
          {competitor.chassisCode || 'N/A'}
        </div>
      </th>
    ))}
  </tr>
</thead>
```

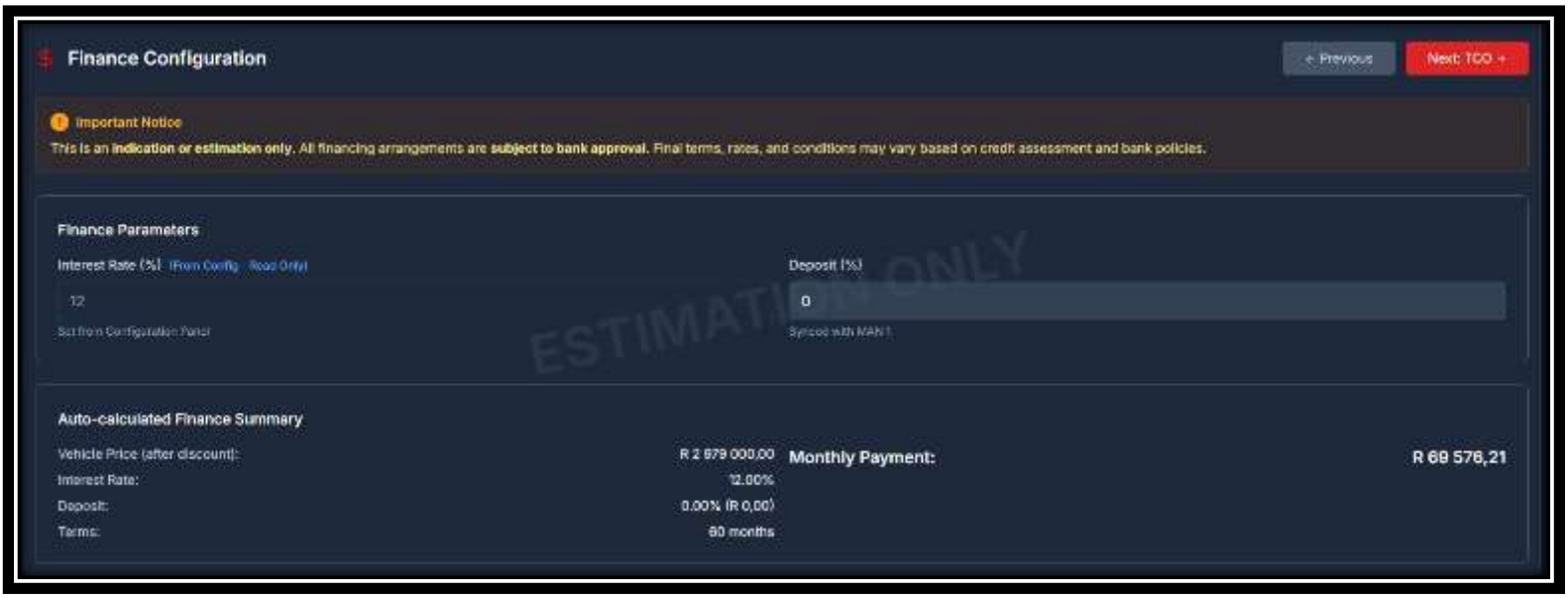
```jsx
{/* Total Price Incl VAT Row - Highlighted */}
<tr className="bg-yellow-100 border-t-2 border-gray-900">
  <td className="border-r border-gray-900 px-3 py-3 text-xs font-bold bg-gray-100">
    Total Price Incl VAT
  </td>
  {/* ✅ MAN Vehicle Columns - Each calculates its own total */}
  {manVehicles.map((manVehicle, manIndex) => (
    <td
      key={`man-total-${manIndex}`}
      className="border-r border-gray-900 px-3 py-3 text-sm font-bold text-center text-man-red-600"
    >
      {formatCurrency(calculateVehicleCost(manVehicle).totalPrice)}
    </td>
  ))}
  {/* Competitor Columns */}
  {competitorCosts.map((costs, index) => (
    <td
      key={`comp-total-${index}`}
      className="border-r border-gray-900 px-3 py-3 text-sm font-bold text-center text-blue-600"
    >
      {formatCurrency(costs?.totalPrice || 0)}
    </td>
  ))}
</tr>
</tbody>
```

TASK 5: Enhanced MFS details with deposit and interest rate display

Added comprehensive financing details to the MFS (MAN Finance Solutions) section of the TCO workflow, displaying deposit percentage, deposit amount, interest rate, and related financing parameters to provide complete visibility into financing configuration and calculations. The implementation required integrating financing parameters from the Finance/MFS workflow step into the final TCO calculator display.

Previously, the MFS section displayed only monthly payment calculations without showing the underlying financing parameters such as deposit percentage, deposit amount, or interest rate. This made it difficult for users to understand how monthly payments were calculated or to verify that financing terms matched customer requirements, as critical financing parameters were hidden from view despite being used in payment calculations.

The deposit display shows both the deposit percentage (typically 20%) and the calculated deposit amount in South African Rands, with the amount calculated as a percentage of the final vehicle price after discounts. The display format uses clear labeling such as "Deposit: 20% (R 150,000.00)" to show both the percentage and absolute value, ensuring users can quickly verify deposit requirements against customer financing arrangements.



The interest rate display presents the annual interest rate percentage used in monthly payment calculations, with clear indication of whether the rate comes from configuration settings (read-only) or from the TCO calculator's MAN 1 vehicle parameters (editable). When the interest rate is sourced from configuration parameters, the display includes a badge indicating "From Config - Read Only" to inform users that they cannot modify the rate directly in the workflow.

The implementation includes validation to ensure deposit percentage and interest rate values are within acceptable ranges (0-100% for deposit, positive values for interest rate) and provides visual feedback when financing parameters affect monthly payment calculations. The display updates dynamically as users modify financing parameters in the Finance/MFS step, providing real-time visibility into how deposit and interest rate changes impact monthly payments and total financing costs.

```
{/* Deposit Percentage Input */}
<div>
  <label className="block text-sm font-medium text-slate-300 mb-2">
    Deposit (%)
  </label>
  <input
    type="number"
    min="0"
    max="100"
    step="0.01"
    value={depositPercentDisplay}
    onChange={(e) => {
      const inputValue = e.target.value;
      setDepositPercentDisplay(inputValue);

      const newDeposit = inputValue === '' ? 0 : (parseFloat(inputValue) || 0);

      // Update TCO calculator data
      if (onTCODataChange && tcoCalculatorData) {
        const currentDeposits = tcoCalculatorData.calculationResults?.manVehicleDepositPercents || {};
        onTCODataChange({
          ...tcoCalculatorData,
          calculationResults: {
            ...tcoCalculatorData.calculationResults,
            manVehicleDepositPercents: {
              ...currentDeposits,
              0: newDeposit // Update MAN 1 deposit percentage
            }
          }
        });
      }
    }}
    className="w-full px-3 py-2 border border-slate-600 rounded-md focus:outline-none focus:ring-2 focus:ring-man-red-50
    placeholder="0.00"
  />
  <p className="text-xs text-slate-400 mt-1">Synced with MAN 1</p>
</div>
```

TASK 6: Universal interest rate configuration (configurable, non-editable in workflow)

Implemented centralized interest rate management through the Configuration Panel, allowing administrators to set a default interest rate that is applied universally across all TCO calculations while preventing workflow users from modifying the rate directly during deal creation. The implementation required creating configuration parameter storage for the default interest rate and modifying the Finance/MFS workflow step to distinguish between configuration-sourced rates and TCO calculator-sourced rates.

Previously, interest rates were either hardcoded in the application (defaulting to 12.25%) or could be freely modified by any user during deal creation, leading to inconsistent interest rate usage across deals and making it difficult to enforce company-wide financing standards. The lack of centralized control meant that different sales personnel might use different interest rates for similar deals, creating pricing inconsistencies and complicating deal comparison and analysis.

The configuration system adds a "default_interest_rate" parameter to the config_parameters table,

stored in the "Finance" category with a default value of 12.25. Administrators can modify this value through the Configuration Panel interface, with changes immediately affecting all new deal creations throughout the system. The configuration value is loaded at application startup and cached in the Finance/MFS component state, ensuring consistent rate application without requiring database queries for every deal.

The read-only enforcement mechanism modifies the Finance/MFS workflow step to detect when the interest rate is sourced from configuration parameters rather than the TCO calculator.

When the rate comes from configuration, the interest rate input field is rendered as disabled with gray background and "From Config - Read Only" badge, preventing workflow users from modifying the centrally-managed rate. This ensures that company-wide financing standards are consistently applied while maintaining transparency about where the rate value originates.

The synchronization system ensures that the configured interest rate is automatically applied to MAN 1 vehicle calculations in the TCO calculator, with the rate value propagating from the Finance/MFS step configuration to the TCO calculator's manVehicleInterestRates array. When administrators update the default interest rate in the Configuration Panel, all new deals automatically use the updated rate, while existing deals retain their originally configured rates to maintain historical accuracy.

```tsx
import { configService } from '../../services/configService';

const FinanceStep: React.FC<FinanceStepProps> = ({ ... }) => {
  // ✅ State for config interest rate
  const [configInterestRate, setConfigInterestRate] = useState<number>(12.25);
  const [configLoaded, setConfigLoaded] = useState<boolean>(false);

  // ✅ Load config interest rate on mount
  useEffect(() => {
    const loadConfigInterestRate = async () => {
      const rate = await configService.getConfigParameterAsNumber(
        'default_interest_rate',
        12.25
      );
      setConfigInterestRate(rate);
      setConfigLoaded(true);
    };
    loadConfigInterestRate();
  }, []);

  // ✅ Determine if using config value (readonly mode)
  const useConfigValue = configLoaded; // If config is loaded, always use config (readonly)

  // ✅ Get interest rate - prioritize config if loaded
  const man1InterestRate = useConfigValue
    ? configInterestRate
    : (tcoCalculatorData?.calculationResults?.manVehicleInterestRates?.[0] ?? configInterestRate);

  // ✅ Sync config value to MAN 1 when config loads
  useEffect(() => {
    // If config is loaded, always use config value (readonly)
    const savedInterestRate = configLoaded
      ? configInterestRate
      : (tcoCalculatorData?.calculationResults?.manVehicleInterestRates?.[0] ?? configInterestRate);

    // Update display
    setInterestRateDisplay(savedInterestRate.toString());
```

```tsx
        // Update TCO calculator data
        if (onTCODataChange && tcoCalculatorData) {
          const currentRates = tcoCalculatorData.calculationResults?.manVehicleInterestRates || {};
          onTCODataChange({
            ...tcoCalculatorData,
            calculationResults: {
              ...tcoCalculatorData.calculationResults,
              manVehicleInterestRates: {
                ...currentRates,
                0: newRate
              }
            }
          });
        }
      }}
      className="w-full px-3 py-2 border border-slate-600 rounded-md focus:outline-none focus:ring-2 focus:ring-man-red-
      placeholder="0.00"
    />
  )}
  {/* ✅ Helper text indicating source */}
  <p className="text-xs text-slate-400 mt-1">
    {useConfigValue ? 'Set from Configuration Panel' : 'Synced with MAN 1'}
  </p>
</div>
```

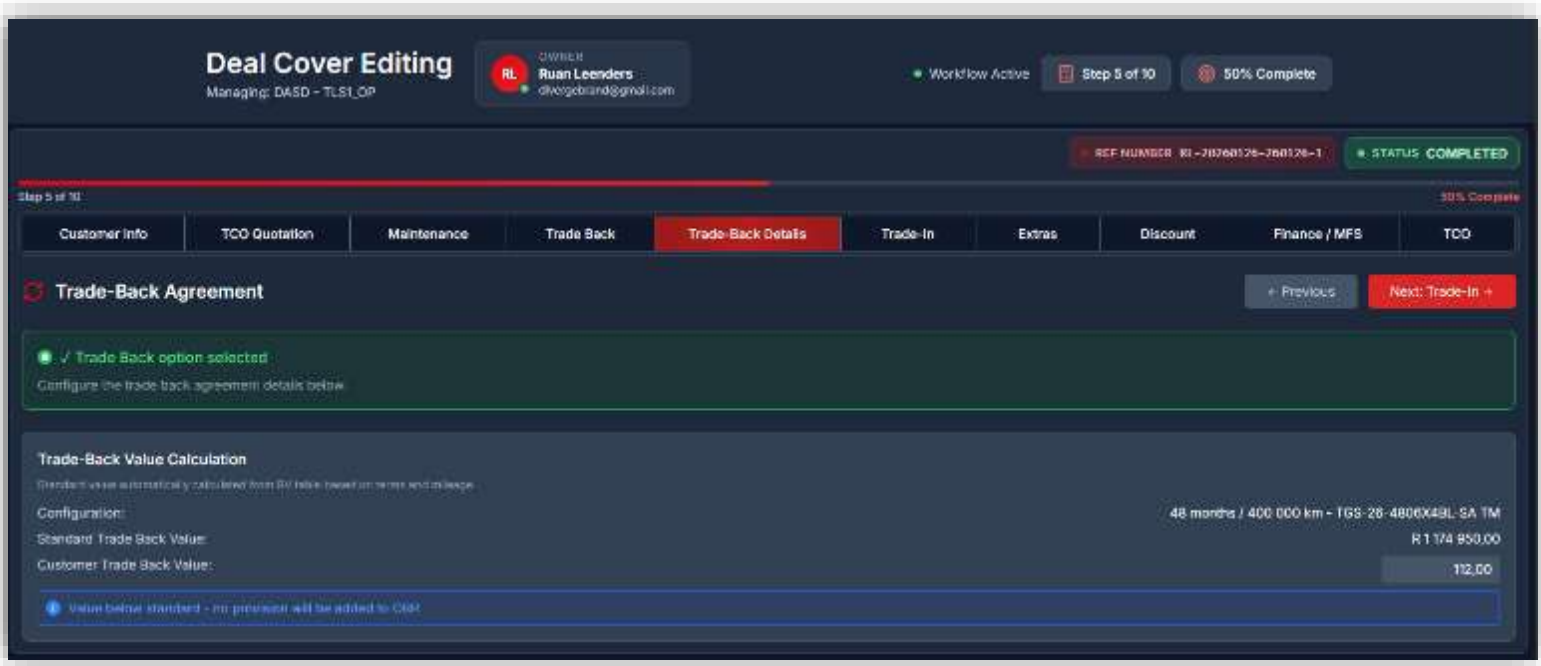
TASK 7: Salesman can decrease trade back value without creating negative provisions

Implemented trade back value adjustment logic that allows sales personnel to decrease the trade back customer value below the standard trade back value without creating negative provisions in the OBR (Offer Book Rate) breakdown. The implementation required modifying the trade back provision calculation formula to use Math.max(0, ...) clamping, ensuring provisions never become negative regardless of how the customer value relates to the standard value.

Previously, when a salesman decreased the trade back customer value below the trade back standard value, the provision calculation (customerValue - standardValue) would produce a negative number, which incorrectly appeared as a negative provision in the OBR breakdown. Negative provisions created accounting confusion because provisions should represent costs or allowances that reduce profitability, not gains that increase profitability. The negative provision also incorrectly affected the total OBR percentage calculation, potentially allowing deals to show artificially low OBR utilization when trade back values were decreased.

The provision calculation fix applies the Math.max(0, customerValue - standardValue) formula to the trade back provision, ensuring that when the customer value is less than or equal to the standard value, the provision is calculated as zero rather than a negative number. This correctly represents the scenario where the customer is receiving a trade back value at or below the standard rate, meaning no additional cost allowance is being provided by the dealership.

The positive provision logic correctly handles the scenario where the customer value exceeds the standard value, calculating the provision as the difference (customerValue - standardValue) to represent the additional cost allowance being provided to secure the deal. This aligns with standard dealership accounting practices where provisions represent costs incurred beyond standard rates to meet customer requirements or win competitive deals.



```
// Calculate individual provision percentages
const extrasProfitPercent = retailPrice > 0
  ? (extrasProfit >= 0 ? -Math.abs(extrasProfit) / retailPrice * 100 : Math.abs(extrasProfit) / retailPrice * 100)
  : 0;
const buyBackAllowancePercent = retailPrice > 0 ? (Math.abs(buyBackAllowance) / retailPrice) * 100 : 0;

// ✅ Trade Back Provision Percentage - Uses absolute value (always positive)
const tradeBackProvisionPercent = retailPrice > 0
  ? (Math.abs(tradeBackProvision) / retailPrice) * 100
  : 0;

const maintenanceProvisionPercent = retailPrice > 0 ? (Math.abs(maintenanceProvision) / retailPrice) * 100 : 0;
const tradeInOverAllowancePercent = retailPrice > 0 ? (Math.abs(tradeInOverAllowance) / retailPrice) * 100 : 0;
const warrantyProvisionPercent = retailPrice > 0 ? (Math.abs(warrantyProvision) / retailPrice) * 100 : 0;

// ✅ Total OBR Amount - Uses absolute value to prevent negative contributions
const totalOBRAmount = Math.abs(extrasProfit) +
                       Math.abs(buyBackAllowance) +
                       Math.abs(tradeBackProvision) + // ✅ Always positive due to Math.max(0, ...)
                       Math.abs(maintenanceProvision) +
                       Math.abs(tradeInOverAllowance) +
                       warrantyContribution;

// ✅ Total OBR Percentage - Sum of all provision percentages
const totalOBRPercentage = extrasProfitPercent +
                           buyBackAllowancePercent +
                           tradeBackProvisionPercent + // ✅ Always positive or zero
                           maintenanceProvisionPercent +
                           tradeInOverAllowancePercent +
                           warrantyProvisionPercent;
```

```
const recalculateOBRBreakdown = () => {
  // ... other provision calculations ...

  // ✅ 3. TRADE BACK PROVISION - Recalculate: customerValue - standardValue (per unit)
  const tradeBackProvision = data.tradeBack?.hasTradeBack
    ? Math.max(0, (data.tradeBack.customerValue || 0) - (data.tradeBack.standardValue || 0))
    : 0;

  // ✅ Generate explanation for trade back allowance
  let tradeBackAllowanceNote = '';
  if (tradeBackProvision > 0 && data.tradeBack?.hasTradeBack) {
    // Positive provision: Customer value exceeds standard value
    const customerValue = data.tradeBack.customerValue || 0;
    const standardValue = data.tradeBack.standardValue || 0;
    tradeBackAllowanceNote = `Customer Trade Back Value - Standard Trade Back Value.\n\nR ${customerValue.toLocaleString
  } else if (data.tradeBack?.hasTradeBack) {
    // ✅ Zero provision: Customer value ≤ standard value
    tradeBackAllowanceNote = 'Trade back selected but allowance is 0 (Customer Value ≤ Standard Value)';
  } else {
    // No trade back selected
    tradeBackAllowanceNote = 'No trade back selected in this deal';
  }

  // ... rest of OBR breakdown calculation ...

  return {
    tradeBackProvision,
    tradeBackAllowanceNote,
    // ... other breakdown values ...
  };
};
```

TASK 8: Increased trade back value creates appropriate provision

Enhanced the trade back provision calculation system to correctly generate positive provisions when sales personnel increase the trade back customer value above the standard trade back value, accurately representing the additional cost allowance being provided to meet customer requirements. The implementation ensures that provision calculations reflect the true cost impact of offering above-standard trade back values to secure deals.

Previously, while the system calculated trade back provisions based on the difference between customer value and standard value, the provision calculation logic and display were not clearly documented or visually represented in the user interface. This made it difficult for sales personnel and managers to understand the cost implications of increasing trade back values above standard rates when negotiating with customers.

The provision calculation implements the formula: tradeBackProvision = customerValue - standardValue (when customerValue > standardValue), which represents the additional allowance being provided beyond the standard trade back rate. This provision is then included in the total OBR (Offer Book Rate) calculation alongside other provisions such as maintenance provision, buy back allowance, and warranty provision, providing a complete picture of all cost allowances being offered in the deal.

The visual representation in the OBR breakdown section displays the trade back provision as a separate line item with the provision amount in South African Rands and the corresponding percentage of retail price. The provision is displayed with an explanatory note that describes the calculation: "Customer Value - Standard Value = Trade Back Allowance", helping users understand how the provision value is derived and what it represents in the context of the deal.

The provision integration into total OBR calculations ensures that when a salesman increases the trade back customer value, the resulting provision correctly reduces the available discount percentage by consuming a portion of the OBR limit. This prevents sales personnel from offering both above-standard trade back values and excessive discounts simultaneously, as both consume the same OBR allowance. The system visually indicates when OBR limits are exceeded due to trade back provisions, with clear messaging about which provisions are contributing to the OBR limit exceedance.

The calculation accuracy is maintained through the provision calculation function that runs whenever trade back values are modified, ensuring real-time updates to provision displays as users adjust customer values or standard values in the Trade Back workflow step. The provision value flows through to quote generation, deal saving, and TCO report generation, ensuring consistent representation of trade back allowances across all deal documentation.

```
// ✅ 3. TRADE-BACK ALLOWANCE = CUSTOMER VALUE - STANDARD VALUE + NEGATIVE EXTRAS
let tradeBackProvision = 0;
if (tradeBack.hasTradeBack) {
  const customerValue = tradeBack.customerValue || 0;
  const standardValue = tradeBack.standardValue || 0; // Standard value from TCO workflow

  // ✅ When customerValue > standardValue: provision = customerValue - standardValue (positive)
  // ✅ When customerValue ≤ standardValue: provision = 0 (no negative provision)
  tradeBackProvision = Math.max(0, customerValue - standardValue);

  // Example scenarios:
  // customerValue = 150000, standardValue = 120000 → provision = 30000 ✅ (positive)
  // customerValue = 120000, standardValue = 120000 → provision = 0 ✅ (zero)
  // customerValue = 100000, standardValue = 120000 → provision = 0 ✅ (not negative)
}

// Add negative extras to trade back allowance
tradeBackProvision += negativeExtrasTotal;
// Keep trade back provision per unit (quantity will be applied in OTP display)
```

```
// ... other provision calculations ...

// ✅ 3. TRADE BACK PROVISION - Calculate: customerValue - standardValue (per unit)
const tradeBackProvision = data.tradeBack?.hasTradeBack
  ? Math.max(0, (data.tradeBack.customerValue || 0) - (data.tradeBack.standardValue || 0))
  : 0;

// ✅ Generate explanation for trade back allowance
let tradeBackAllowanceNote = '';
if (tradeBackProvision > 0 && data.tradeBack?.hasTradeBack) {
  // ✅ POSITIVE PROVISION: Customer value exceeds standard value
  const customerValue = data.tradeBack.customerValue || 0;
  const standardValue = data.tradeBack.standardValue || 0;

  // ✅ Clear formula explanation showing the calculation
  tradeBackAllowanceNote = `Customer Trade Back Value - Standard Trade Back Value.\n\nR ${customerValue.toLocaleString
} else if (data.tradeBack?.hasTradeBack) {
  // Zero provision: Customer value ≤ standard value
  tradeBackAllowanceNote = 'Trade back selected but allowance is 0 (Customer Value ≤ Standard Value)';
} else {
  // No trade back selected
  tradeBackAllowanceNote = 'No trade back selected in this deal';
}

return {
  tradeBackProvision, // ✅ Positive value when customerValue > standardValue
  tradeBackAllowanceNote, // ✅ Explanation of calculation
  // ... other breakdown values ...
};
```

TASK 9: Decreased trade back value does not affect provisions (no negative provisions)

Implemented protection against negative provision creation when trade back customer values are decreased below standard trade back values, ensuring that OBR calculations remain accurate and provisions correctly represent cost allowances rather than gains. The implementation builds on the Math.max(0, ...) clamping logic to ensure provisions are always zero or positive values.

Previously, the provision calculation formula (customerValue - standardValue) would produce negative values when the customer value was set below the standard value, such as when a salesman negotiated a trade back value of R100,000 while the standard trade back value was R120,000. This negative provision of -R20,000 would incorrectly reduce the total OBR percentage, making it appear that the deal was consuming less OBR allowance than it actually was, and potentially allowing excessive discounts to be approved.

The negative provision prevention ensures that when a salesman decreases the trade back customer value, the provision calculation returns zero rather than a negative number, correctly representing that no additional cost allowance is being provided beyond standard rates. This aligns with dealership accounting principles where provisions represent costs incurred above standard rates, not savings realized by offering below-standard rates.

The OBR impact is correctly calculated by excluding negative provisions from the total OBR percentage, ensuring that decreased trade back values do not artificially reduce OBR utilization or increase available discount percentage. The system maintains accurate OBR tracking regardless of whether trade back values are increased above standard (creating positive provisions) or decreased below standard (creating zero provisions), providing consistent deal analysis across all trade back scenarios.

The provision display logic includes explanatory notes in the OBR breakdown section that indicate when trade back provision is zero due to the customer value being at or below the standard value. This provides transparency into the provision calculation logic and helps users understand why provision values may be zero even when trade back is enabled, supporting better understanding of deal cost structures and OBR utilization patterns.

TASK 10: Stand-in value equals offer to customer (no credit given on loading)

Implemented automatic stand-in value initialization that sets the stand-in value equal to the offer to customer value when trade-in vehicles are loaded, ensuring that trade-in over-allowance calculations start at zero until the user explicitly modifies the stand-in value to reflect the actual vehicle valuation. The implementation prevents incorrect credit calculations that would occur if stand-in values defaulted to different amounts than the offer to customer values.

Previously, when trade-in vehicles were loaded into the system, the stand-in value might default to zero or a different value than the offer to customer value, creating an immediate over-allowance calculation that did not reflect the actual trade-in arrangement. This caused confusion in OBR calculations because the trade-in over-allowance provision would show non-zero values before the user had properly assessed the vehicle's stand-in value, making it appear that additional allowances were being provided when they were not.

The initialization logic sets standInValue = amountOfferedToCustomerExclVat when trade-in vehicles are loaded from the database or created in the workflow, ensuring that the over-allowance calculation (amountOffered - standInValue) starts at zero. This correctly represents that no over-allowance is being provided until the user explicitly sets a different stand-in value based on vehicle valuation, market assessment, or dealer trade-in policies.

The over-allowance calculation only becomes non-zero when the user modifies the stand-in value to differ from the offer to customer value, such as when a vehicle is assessed at R80,000 stand-in value but the dealership offers the customer R90,000, creating a R10,000 over-allowance that is correctly reflected as a provision in the OBR breakdown. This over-allowance provision then correctly consumes OBR allowance, reducing available discount percentage and ensuring accurate deal profitability tracking.

The UI implementation includes clear messaging in the Trade-In Management interface that indicates the relationship between offer to customer, stand-in value, and over-allowance. The interface displays all three values side-by-side for each trade-in vehicle, with the over-allowance automatically calculated and highlighted when it exceeds zero, providing immediate visibility into trade-in allowances being provided.

The trade-in provision integration ensures that total trade-in over-allowance (sum of all trade-in vehicle over-allowances divided by deal quantity) flows correctly into the OBR breakdown as a separate provision line item, with appropriate percentage calculation relative to retail price. This provision correctly contributes to total OBR utilization alongside other provisions such as trade back provision, maintenance provision, and buy back allowance, providing a complete picture of all cost allowances being offered in the deal.

The implementation includes proper handling of multiple trade-in vehicles, calculating total over-allowance across all trade-ins and dividing by deal quantity to determine per-unit provision impact. This ensures that OBR calculations remain accurate whether a deal includes one trade-in vehicle or multiple trade-in vehicles, with each trade-in's over-allowance contributing appropriately to the total deal cost allowance.

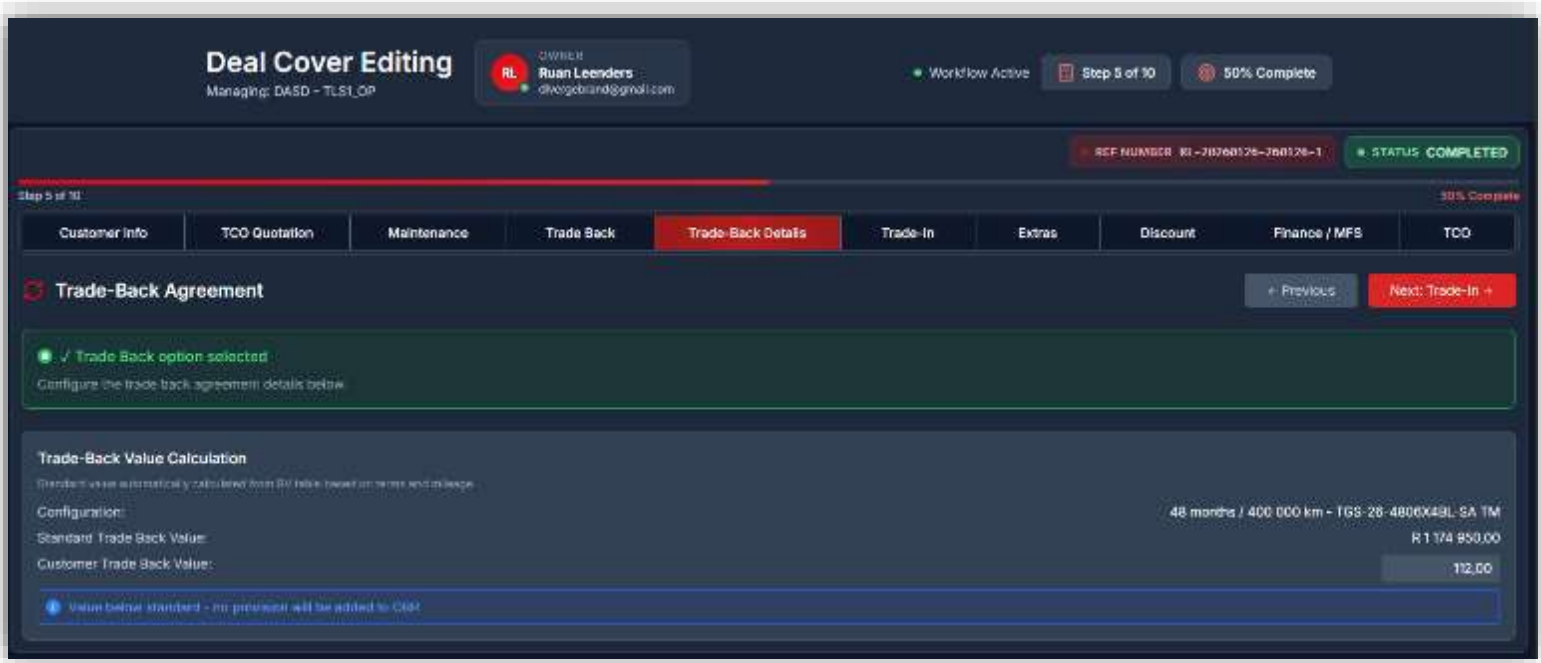
TASK 11: Added UI user messages and guidance to Trade Back workflow step

Implemented comprehensive user interface messaging and guidance throughout the Trade Back workflow step to help users understand trade back concepts, calculation logic, and the relationship between standard values, customer values, and provisions. The implementation required adding informational panels, tooltips, and explanatory notes at key points in the Trade Back interface to improve user comprehension and reduce errors in trade back configuration.

Previously, the Trade Back workflow step presented input fields for standard value and customer value without explaining what these values represented, how they were used in calculations, or what impact they had on deal profitability. This lack of guidance caused confusion among sales personnel, particularly new users who were unfamiliar with trade back concepts and provision calculations, leading to incorrectly configured trade back values that created inaccurate quotes and deal documentation.

The informational panel implementation adds a blue-themed notice box at the top of the Trade Back step that explains the purpose of trade back, defines standard value and customer value, and describes how the provision is calculated. The panel uses clear, non-technical language to explain that

standard value represents the baseline trade back rate based on company policy or market conditions, while customer value represents the actual trade back amount being offered to the customer to secure the deal.



The provision calculation explanation includes a visual formula display showing "Provision = Customer Value - Standard Value (when Customer Value > Standard Value)" with color-coded examples demonstrating both positive provision scenarios (customer value exceeds standard value) and zero provision scenarios (customer value equals or is below standard value). This visual learning aid helps users understand how their trade back value entries affect deal profitability and OBR utilization.

The validation messaging system provides real-time feedback when users enter trade back values that create significant provisions, warning users when the trade back provision exceeds certain thresholds (such as 5% of retail price or 10% of retail price) that may require manager approval. The warning messages use amber-themed styling to draw attention without blocking the user from proceeding, supporting informed decision-making while maintaining workflow flexibility.

The limits and constraints section displays the relationship between trade back values and balloon payments, explaining that balloon payment cannot exceed the trade back customer value and providing clear guidance on acceptable value ranges. When users attempt to configure trade back values that conflict with balloon payment settings or exceed vehicle price limits, contextual error messages explain the constraint and suggest corrective actions, reducing trial-and-error configuration attempts.

The tooltips system adds hover-based help text to complex fields such as "Standard Value" and "Customer Value", providing brief definitions and examples directly in the interface without requiring users to navigate away from the workflow to access help documentation. The tooltips use dark-themed styling consistent with the application's design language and appear instantly on hover, making help information readily accessible throughout the trade back configuration process.

The implementation significantly improves the user experience for trade back configuration by transforming the Trade Back step from a form with unlabeled input fields into an educational interface that guides users through proper trade back setup while explaining the business logic and calculation impacts of their configuration choices. The comprehensive messaging reduces training requirements for new users while supporting experienced users with quick-reference information about trade back calculation rules and constraints.

```
<div className="bg-blue-900/20 border border-blue-600 rounded-lg p-4 mb-6">
  <div className="flex items-start space-x-3">
    <div className="w-5 h-5 bg-blue-500 rounded-full flex items-center justify-center flex-shrink-0 mt-0.5">
      <span className="text-white text-xs font-bold">i</span>
    </div>
    <div className="flex-1">
      <h3 className="text-blue-300 font-semibold mb-2">Understanding Trade Back</h3>
      <div className="text-sm text-blue-200 space-y-2">
        <p>
          <strong>Trade Back</strong> allows customers to return their vehicle at the end of the contract term
          for a guaranteed value. This value can be used as a balloon payment or trade-in credit.
        </p>
        <div className="mt-3 space-y-1">
          <p>
            <strong className="text-blue-300">Standard Value:</strong> The baseline trade back rate based on
            company policy or market conditions. This is automatically calculated from the RV table based on
            your vehicle configuration (terms, mileage, and deployment type).
          </p>
          <p>
            <strong className="text-blue-300">Customer Value:</strong> The actual trade back amount being
            offered to the customer to secure the deal. You can increase this above the standard value to
            provide additional incentive, or decrease it if needed.
          </p>
        </div>
      </div>
    </div>
  </div>
</div>
```

PROJECT 2: MAN CONTRACT MANAGEMENT

Project Overview and Business Context

This Contract Management System is a full-stack web application for managing business contracts from creation through execution. It supports contract creation, approval workflows, digital signatures, document management, and audit trails. The system handles Standard and Non-Standard contracts with different workflows, manages internal and external signatories, and provides role-based access control for Admin, Manager, Legal, and User roles.

Technology Stack and Architecture

The frontend uses React 19 with TypeScript for type safety. The UI is built with Tailwind CSS 4.1.17. Vite 7.2.4 handles development and builds. React Router 7.10.1 manages navigation. Lucide React provides icons. PDF handling uses pdf-lib, html2canvas, jspdf, and jspdf-autotable. Excel export uses xlsx.The backend uses Supabase as a PostgreSQL-based BaaS. The database includes JSONB for flexible data, UUID primary keys, and Row Level Security (RLS) for access control. Supabase Edge Functions provide serverless API endpoints. Authentication uses Supabase Auth with JWT tokens. Docker runs the local Supabase development environment.

Database Schema and Migrations

The database schema was built through 13 migrations. Migration 001 created core tables: contracts, tasks, signatories, users, and audit_logs. Migration 002 added seed data. Migration 003 fixed RLS policies. Migration 004 added contract_value. Migration 005 added templates. Migration 006 set up Supabase Storage. Migrations 007 and 008 fixed admin policies. Migration 009 added signature columns. Migration 010 created repository tables. Migration 011 fixed tasks and signatories relationships. Migration 012 created signature_tokens for external signing. Migration 013 fixed missing INSERT policies for tasks, which was blocking task creation.

Task 1: Contract Creation Workflow

The contract creation workflow is a 5-step wizard in the NewContract component. Step 1 collects contract details: contract number, title, description, type, classification (Standard or Non-Standard), counterparty type, vendor or customer name, department, business unit, and contract owner. The classification determines whether legal review is required. Step 2 handles document upload or selection from the repository, validates PDFs, and stores them in Supabase Storage. Step 3 configures signatures: external signatories with name, email, title, signature method, due date, reminder frequency, and escalation contact; internal signatories with role, name, order, title, signing mode (sequential or parallel), final approver, and SLA per step. Step 4 sets lifecycle dates: effective date, expiry date, renewal notice date, and destruction date, with validation. Step 5 shows a workflow preview before submission. On submit, the contractService creates the contract record, uploads documents, creates signatory records, and generates tasks based on classification and workflow configuration.

Task 2: Workflow Stage Management

The WorkflowManager component visualizes and manages contract stages: Draft, Pending, Legal Review, Sent, Signed, Completed, and Rejected. The system transitions contracts through these stages based on task completions. Standard contracts skip legal review and go from Pending to Sent. Non-Standard contracts require legal review before proceeding. The component displays the current stage, next stages, and completion status with color-coded indicators. Stage transitions are triggered by task approvals and signature completions.

Task 3: Task Management System

The task management system automatically creates tasks when contracts are submitted. Legal review tasks are created for Non-Standard contracts and assigned to Legal role users. External signature tasks are created for each external signatory and assigned to the contract creator for monitoring. Internal signature tasks are created for each internal signatory in the specified order. Tasks have statuses: pending, in_progress, approved, and rejected. The ApprovalsPage component displays tasks in tabs by status. Tasks show contract details, action type, priority, due date, and allow approve or reject actions. The system auto-refreshes every 30 seconds and on window focus to catch external signatures. The taskService handles task creation, retrieval, updates, and completion. A critical fix was applied in migration 013 to add INSERT RLS policies, which were preventing task creation.

Task 4: Digital Signature Implementation

The DigitalSignature component supports three signature methods. The draw method uses an HTML5 canvas for freehand drawing with touch and mouse support, undo/redo, and clear. The type method generates a stylized signature from typed text using canvas text rendering. The upload method allows image upload with validation and preview. All methods output base64-encoded PNG images. The component includes signature preview, confirmation, and validation before submission.

Task 5: Internal Signature Workflow

Internal signatures are handled through the ApprovalsPage. When a user approves an internal signature task, a signature modal opens with the DigitalSignature component. After capture, the taskService.approveTaskWithSignature method saves the signature image to the signature_proof JSONB field in the signatories table, updates the task status to approved, records completion timestamp, and creates the next task in the sequence if using sequential signing. The signature image is displayed in contract details and in the completed tasks view.

Task 6: External Signature Workflow

External signatures use a token-based system. When an external signature task is approved, the signingService.createSigningToken method generates a secure 64-character hexadecimal token, creates a signature_tokens record with contract ID, task ID, signatory details, and expiration date, and returns a signing URL. The ExternalSigningPage component validates the token, checks expiration and usage status, displays contract details, and provides the DigitalSignature component for signature capture. After signing, the signingService.completeExternalSignature method updates the token with signature data, marks it as used, updates the task status, updates the contract workflow stage, creates an internal signature task if needed, and saves the signature to the signatories table with full proof including image, method, timestamp, and IP address. A critical bug fix was applied where the signature image was not being saved to the signatories table for external signatures, which was corrected by including the signatureImage in the signature_proof JSONB field.

Task 7: Contract Management and Viewing

The ContractManagement component provides a comprehensive contract listing with filtering by status, type, department, and search functionality. The contract details view shows all contract metadata, workflow stage visualization, document preview and download, signatory list with signature status, signature images for completed signatures, and audit trail. The component uses the contractService to fetch contracts with related data including signatories and documents. Signature images are displayed from the signature_proof JSONB field, showing the actual signature image, signer name, method used, and timestamp.

Task 8: User Authentication and Authorization

The authentication system uses Supabase Auth with email and password. The UnifiedAuthForm component handles sign-in and sign-up with validation. The AuthContext provides authentication state management throughout the application. The ProtectedRoute component ensures only authenticated users can access protected pages. User roles include Admin, Manager, Legal, and User, with different permissions. The WaitingForApproval component handles users pending admin approval. The system includes email verification, password reset, and MFA setup components. RLS policies enforce role-based access at the database level.

Task 9: Admin Panel Features

The admin panel includes UserManagement for creating, editing, and approving users, with role assignment and status management. CustomerManagement handles counterparty information. SystemLogs displays audit trails and system events. ConfigurationPanel provides system settings and configuration options. All admin features use the userService and auditService to interact with the database.

Task 10: Document Management

The documentService handles document uploads to Supabase Storage, document retrieval by contract ID, document versioning, and document deletion. The FileManager component provides a UI for browsing and managing documents. Documents are stored in organized buckets with metadata stored in the contract_documents table. The system supports PDF preview and download functionality.

Task 11: Reporting and Export

The ContractReports component provides contract analytics including status distribution, contract value summaries, expiration tracking, and signature status reports. The exportService uses jspdf and xlsx to generate PDF and Excel exports of contract data, reports, and audit logs. The ReportsPage component displays these reports with filtering and export options.

Task 12: Repository and Templates

The RepositoryPage component provides a document repository for storing and retrieving contract templates and archived documents. The TemplatesPage component manages contract templates that can be used when creating new contracts. The templateService handles template CRUD operations. Templates include pre-configured fields, workflows, and signature configurations.

Task 13: Bug Fixes and Critical Issues Resolved

Several critical bugs were identified and fixed during development. The task creation failure was caused by missing INSERT RLS policies on the tasks table, resolved in migration 013. The signature image not displaying issue was caused by the signingService not saving the signatureImage to the signatories table for external signatures, fixed by updating the completeExternalSignature method. A SQL backfill script was created to sync existing signature images from signature_tokens to signatories. The heading background color issue was caused by a global CSS rule applying background-color to all h1-h6 elements, fixed by setting background-color to transparent. External signature links were missing because signature_tokens were not being created, resolved by ensuring token creation in the approval workflow.

Task 14: Database Operations and Testing

Extensive database operations were performed including creating test users with proper authentication entries in both auth.users and public.users tables, creating test contracts with all required fields including contract_classification, department, business_unit, contract_owner, renewal_notice_date, and destruction_date, creating test tasks for legal review, external signature, and internal signature, creating signature tokens for external signing links, and backfilling signature data for existing signed contracts. All operations used Docker exec commands to interact with the PostgreSQL container, copying SQL files into the container and executing them with psql.

Task 15: UI and Styling Improvements

The application uses a consistent dark theme with Tailwind CSS. The Sidebar component provides navigation with role-based menu items. The Dashboard component shows contract statistics and recent activity. Error boundaries handle React errors gracefully. The index.css file contains global styles with CSS custom properties for colors, spacing, and typography. The heading background color issue was resolved by modifying global CSS rules. All components use responsive design principles and consistent spacing and color schemes.

Development Process and Methodologies

The development process involved iterative debugging, identifying issues through user testing, tracing problems through the codebase using codebase search and grep tools, fixing issues at the source, and verifying fixes through database queries and UI testing. Database migrations were used for schema changes, ensuring version control and reproducibility. The local development environment used

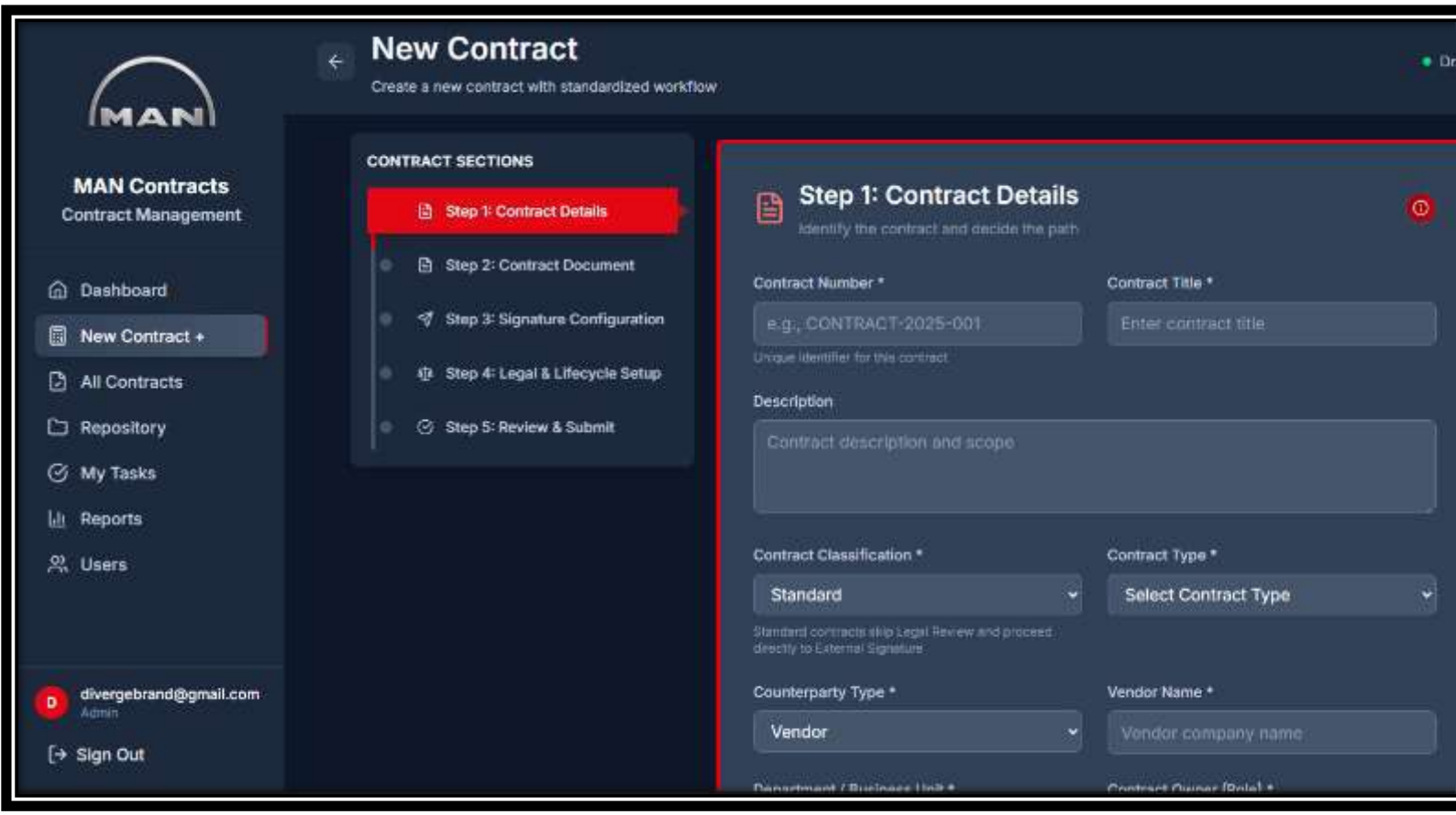
Docker for Supabase services, requiring knowledge of Docker commands and container management. TypeScript provided type safety throughout the codebase, catching errors at compile time. The service layer pattern separated business logic from UI components, making the codebase maintainable and testable.

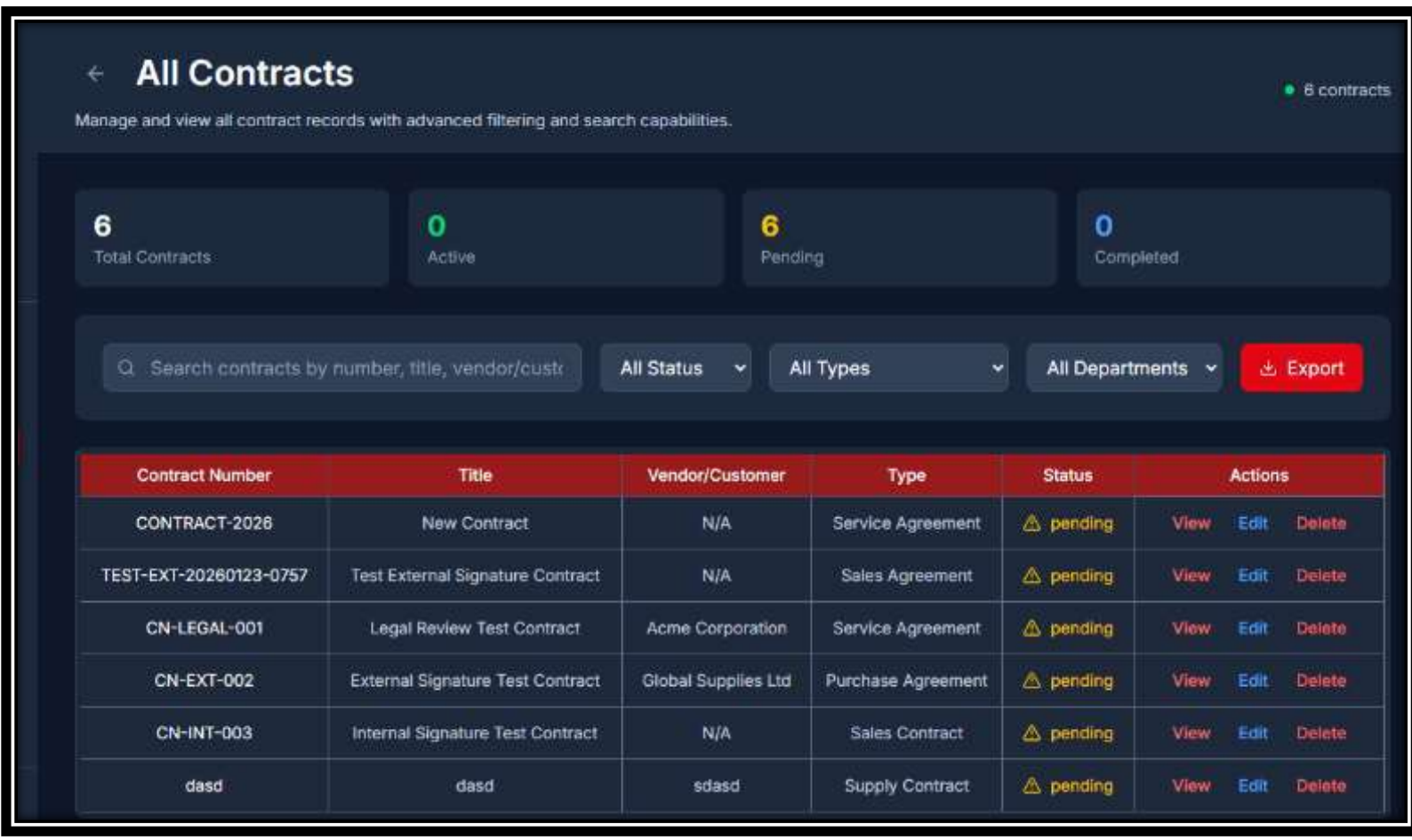
Key Achievements and Deliverables

The system successfully implements a complete contract lifecycle management workflow with automated task generation, digital signature capture and storage, external signing via secure token-based links, role-based access control with RLS policies, comprehensive audit trails, document management with versioning, reporting and export capabilities, and a responsive user interface. All critical bugs were identified and resolved, the database schema was properly designed with 13 migrations, and the system is ready for production deployment with proper error handling and validation throughout.

Technical Skills Demonstrated

This project demonstrates proficiency in React and TypeScript for building complex user interfaces, PostgreSQL database design with advanced features like JSONB and RLS, Supabase platform integration including Auth, Database, Storage, and Edge Functions, Docker container management for local development, SQL query writing and database administration, debugging complex full-stack applications, working with PDF generation and manipulation libraries, implementing secure authentication and authorization, managing state in React applications, and following software engineering best practices including service layer patterns, error handling, and code organization.


New Contract
Create a new contract with standardized workflow
MAN Contracts
Contract Management
Dashboard
New Contract +
All Contracts
Repository
My Tasks
Reports
Users
divergebrand@gmail.com
Admin
Sign Out
CONTRACT SECTIONS
Step 1: Contract Details
Step 2: Contract Document
Step 3: Signature Configuration
Step 4: Legal & Lifecycle Setup
Step 5: Review & Submit
Step 1: Contract Details
Identify the contract and decide the path
Contract Number *
e.g., CONTRACT-2025-001
Unique identifier for this contract
Contract Title *
Enter contract title
Description
Contract description and scope
Contract Classification *
Standard
Standard contracts skip Legal Review and proceed directly to External Signature
Contract Type *
Select Contract Type
Counterparty Type *
Vendor
Vendor Name *
Vendor company name


All Contracts
Manage and view all contract records with advanced filtering and search capabilities.
6 contracts
6
Total Contracts
0
Active
6
Pending
0
Completed
Search contracts by number, title, vendor/cust
All Status
All Types
All Departments
Export
Contract Number
Title
Vendor/Customer
Type
Status
Actions
CONTRACT-2026
New Contract
N/A
Service Agreement
pending
View Edit Delete
TEST-EXT-20260123-0757
Test External Signature Contract
N/A
Sales Agreement
pending
View Edit Delete
CN-LEGAL-001
Legal Review Test Contract
Acme Corporation
Service Agreement
pending
View Edit Delete
CN-EXT-002
External Signature Test Contract
Global Supplies Ltd
Purchase Agreement
pending
View Edit Delete
CN-INT-003
Internal Signature Test Contract
N/A
Sales Contract
pending
View Edit Delete
dasd
dasd
sdasd
Supply Contract
pending
View Edit Delete

NEIGHBORHOOD SECURITY APP

What we do

Neighbourhood Bot is a smart security platform that helps residential communities monitor who enters their neighbourhood. When a vehicle passes a gate or camera, the system reads the licence plate, analyses the vehicle and driver, and assigns a risk level. Residents and admins get instant alerts on their phones so they can decide whether the visitor is welcome or needs attention. The goal is to give neighbourhoods better visibility and control over who comes and goes, without turning every street into a fortress.

The problem we solve

Residential estates and gated communities often rely on manual guards, basic cameras, or nothing at all. Unknown vehicles enter and leave with no record, no context, and no way for residents to react quickly. When something goes wrong—a suspicious car, a repeat offender, or a wanted vehicle—communities find out too late. Neighbourhood Bot turns existing gate cameras into an intelligent layer that works 24/7, giving residents peace of mind and admins the tools to act fast.

How it works

Cameras at entry points send licence-plate events to our backend. Our system runs checks: it identifies the plate, compares it against blacklists and known vehicles, and uses AI to analyse the vehicle and driver. Each event gets a risk score—green for familiar and safe, yellow for needs review, red for high priority. Admins can vote on yellow alerts (friend or foe), and the community builds a shared understanding of who belongs. Residents see live alerts, recent activity, and can manage their own household's vehicles and guests. The flow is designed to be fast, clear, and actionable.

Technology and build

The product is built as a modern, scalable stack. The mobile app runs on Flutter, so it works on both Android and iOS from a single codebase. The backend is written in Python using FastAPI, a high-performance framework suited for real-time APIs. We use Supabase for authentication, database, and storage, which keeps infrastructure costs low while supporting growth. AI services handle licence-plate recognition, vehicle analysis, and face detection, with a risk engine that combines multiple signals into a single, auditable score. The system integrates with industry-standard cameras (such as Hikvision) via webhooks, so estates can plug in what they already have.

Languages and codebase

The frontend is written in Dart (Flutter), and the backend in Python. We use TypeScript for small utility scripts (e.g. email delivery). The codebase is structured for clarity: screens, services, and providers are separated so new features can be added without touching unrelated code. We favour readable, maintainable code over clever shortcuts, which makes it easier to onboard developers and scale the team.

Why it matters

Communities want to feel safe without feeling locked in. Neighbourhood Bot delivers that: smarter use of existing cameras, instant visibility, and a shared sense of control. We are building a platform

that can grow from a single estate to many, with a clear path from pilot to paid deployment. Our focus is on reliability, simplicity, and trust—the foundations investors and customers both care about.

Feature 1: Snapshot image on alert detailDescription

When an alert has a camera snapshot, the app shows that image at the top of the Alert Detail screen so the user can see the evidence (vehicle/capture) before the metadata and actions.

Where it appears

- **Screen:** Alert Detail (opened from Alerts tab when you tap an alert).
- **Position:** Full-width image at the top, above the status badge and the rest of the details (plate, score, "Why flagged", voting, etc.).
- **When:** Only if that alert has a snapshot_uri in the database; otherwise no image is shown.

Database

- **Table:** alerts
- **Column:** snapshot_uri (TEXT, nullable)
- **Source:** Set by the backend when the webhook creates an alert: image is downloaded from the camera, uploaded to Supabase Storage, and the public URL is stored in alerts.snapshot_uri.
- **Related:** Alert row is created from a vehicle_events row (camera event); alerts.event_id references vehicle_events.id.

How it works (flow)

1. Camera sends event → webhook receives it.
2. Backend downloads the image, uploads to Supabase Storage (snapshots/...), gets a public URL.
3. Backend inserts into alerts with snapshot_uri set to that URL.
4. App loads alerts from Supabase (including snapshot_uri).
5. On Alert Detail, if alert.imageUri (mapped from snapshot_uri) is non-empty, the app shows an image widget that loads that URL, with loading and error handling.

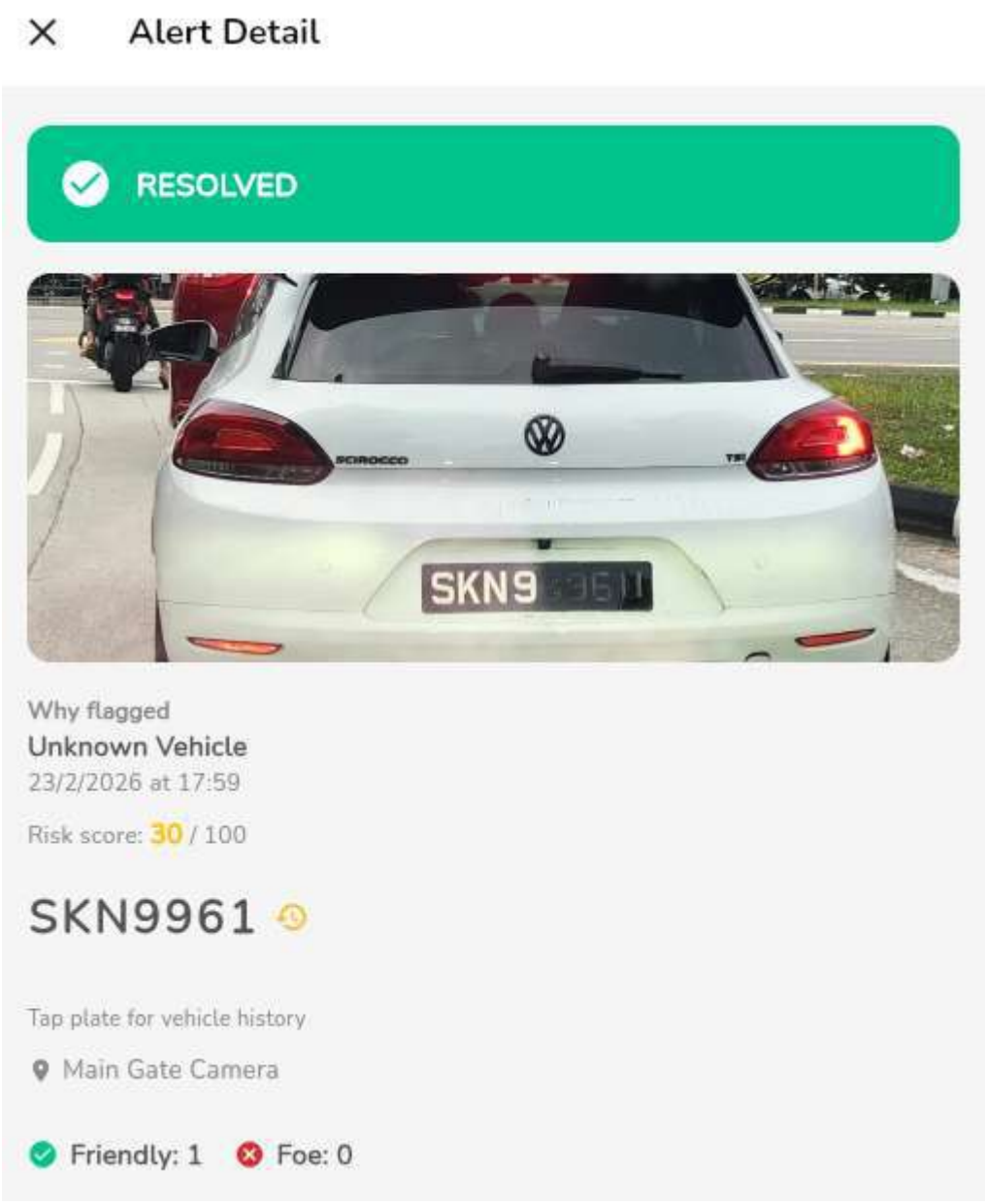
Code

- **Frontend:** alert_detail_screen.dart — conditional block that, when alert.imageUri is set, renders an Image.network with loading and error placeholders.
- **Model:** Alert in models.dart has imageUri; Alert.fromSupabase maps row['snapshot_uri'] to i mageUri.

Tested / status

- **Tested:** Yes.
- **Result:** Snapshot is displayed on Alert Detail for alerts that have snapshot_uri in the DB (e.g. camera-created alerts such as CA145747, TN13563, SKN9961, LD98DJGP).

- **Note:** Alerts without snapshot_uri (e.g. manually created or from other flows) correctly show no image; the rest of the detail screen still works



Feature 2: Vehicle profile (score trend + prior alerts)

Description

When a user taps the plate on an alert detail screen, the app opens a **Vehicle profile** screen. It shows the plate, vehicle description (if available), and a list of all prior alerts for that plate, each with risk score, status, entry point, and time. Tapping a prior alert opens its alert detail screen.

Where it appears

- **Entry:** Alert Detail screen → tap the plate chip (car icon + plate + "History" pill).
- **Screen:** Vehicle Profile (vehicle_profile_screen.dart).
- **Content:** Plate and vehicle description at the top; "Score trend & prior alerts" list below; each row is tappable and opens that alert's detail.

Database

- **Table:** alerts
- **Columns**
  used: id, plate, vehicle_make, vehicle_color, status, risk_score, entry_point, created_at (or timestamp)
- **Logic:** No extra table. The app filters app.alerts by plate (case-insensitive) and sorts by timestamp descending. All data comes from the alerts already loaded in memory.

How it works (flow)

1. User opens an alert from the Alerts tab.
2. On Alert Detail, the plate is shown in a tappable chip.
3. User taps the plate → Navigator.push(VehicleProfileScreen(plate, vehicleMake, vehicleColor)).
4. Vehicle Profile receives the plate and uses Consumer<AppProvider> to read app.alerts.
5. It filters alerts where plate.trim().toUpperCase() == normalizedPlate.
6. Results are sorted by timestamp descending.
7. Each row shows score, status, entry point, and time ago; tapping a row opens AlertDetailScreen(alertId).

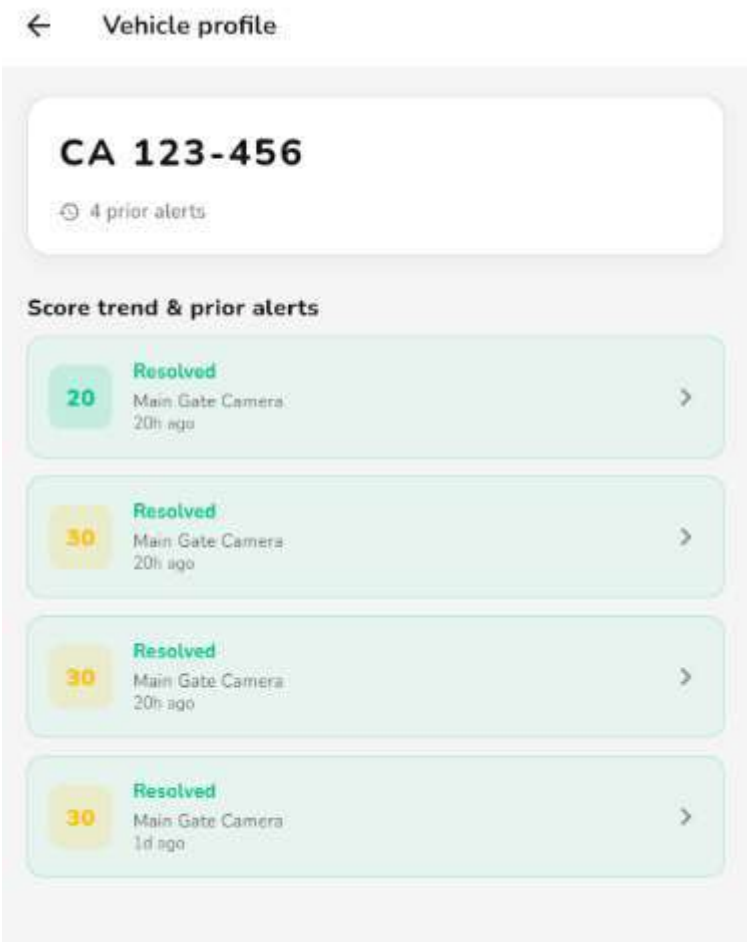
Code

- **Screen:** vehicle_profile_screen.dart – StatelessWidget, Consumer<AppProvider>, filters alerts by plate, builds list.
- **Entry:** alert_detail_screen.dart – plate chip InkWell that pushes VehicleProfileScreen with plate, vehicleMake, vehicleColor.

- **Data:** Uses app.alerts from AppProvider (loaded from Supabase alerts table).

Tested / status

- **Tested:** Yes.
- **Result:** Vehicle profile opens when tapping the plate; shows the correct prior alerts for that plate (e.g. CA 123-456 shows 4 resolved alerts); tapping a prior alert opens its detail screen; back navigation works.
- **Note:** If there are no prior alerts for a plate, the screen shows “No prior alerts for this plate.”



Feature 3: Neighbourhood Admin – Manage Members (search, filters, household dropdown)

Description

Neighbourhood admins can manage all members in their neighbourhood from a single screen. Members are grouped by household in expandable sections, with search and filters (Status, Role, Doc, Household). House admins use a dark card style; regular members use a yellow card. Each member card has a 3-dot menu for Deactivate/Activate. Invitees (members added by household admins) are included even if their profile has no neighbourhood set.

Where it appears

- **Entry:** Admin Dashboard → "Manage all members".
- **Screen:** Manage Members (neighbourhood_members_screen.dart).
- **Layout:** Search bar at top; filter chips (Status, Role, Doc, Household); Open all / Close all; expandable household sections; member cards inside each section.
- **Member cards:** House Admin = dark glassy card with ADMIN badge; House Member = yellow card with doc status chip; 3-dot menu for Deactivate/Activate (except for the current user).

Database

- **Tables:** profiles, households, household_members
- **Profiles columns:** id, full_name, email, phone, role, doc_status, onboarded, status, neighbourhood
- **Logic:** Members are loaded from:

1. Profiles with neighbourhood set.
2. Profiles linked via household_members for households in the neighbourhood (includes invitees without neighbourhood).

- **Migration 020:** add_household_member sets neighbourhood on new profiles; RLS allows neighbourhood admins to read profiles of members in neighbourhood households.

How it works (flow)

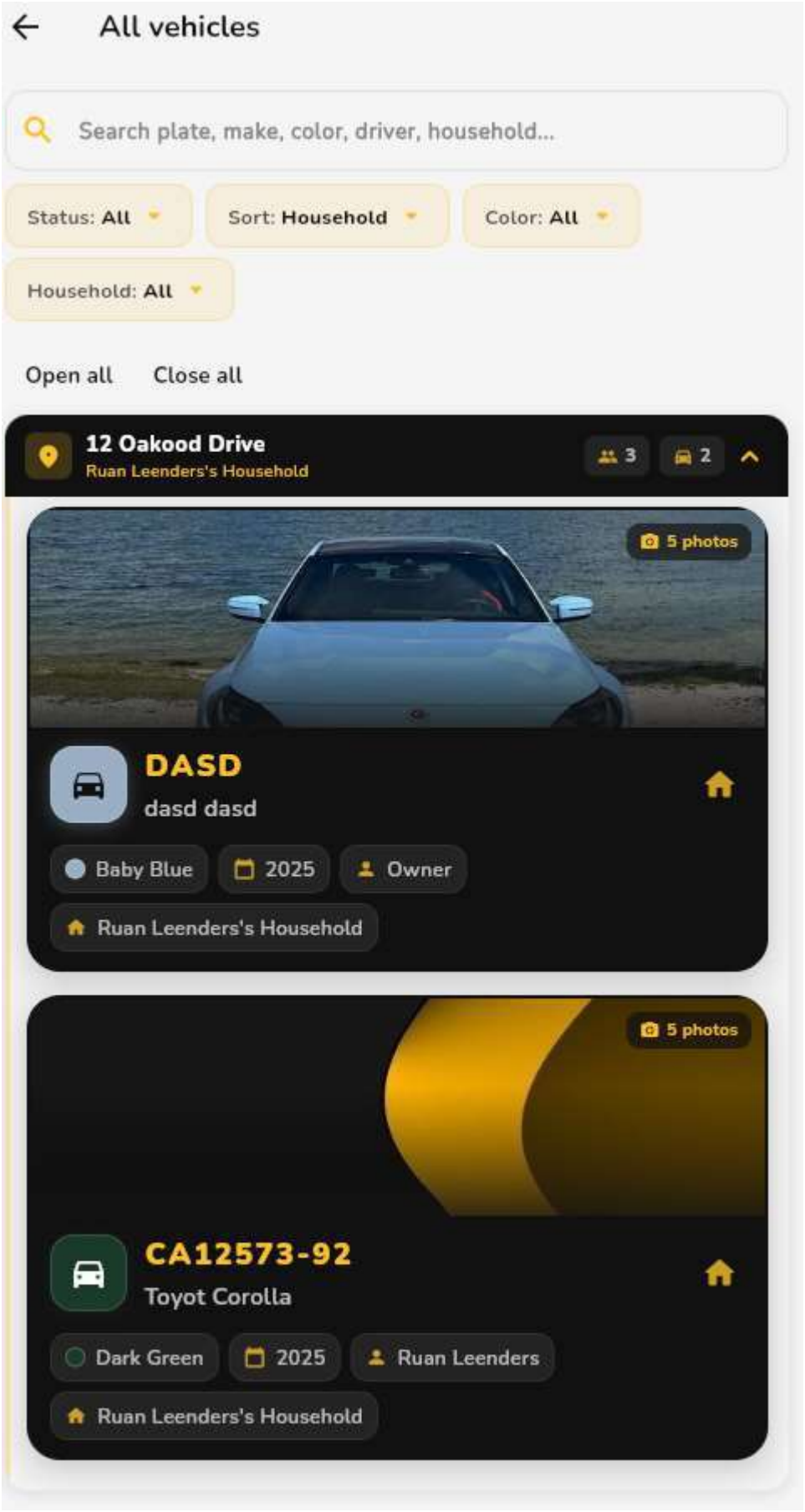
1. User opens Admin Dashboard and taps "Manage all members".
2. App loads members and households from Supabase.
3. Members are grouped by household_id (from household_members).
4. Search filters by name, email, phone; filters by Status, Role, Doc, Household.
5. Each household is shown as a header (address, name, member count) with expand/collapse.
6. Tapping a header expands/collapse the member list.
7. Member cards show role (House Admin vs House Member), contact info, doc status.
8. 3-dot menu on each card (except self) offers Deactivate or Activate.
9. Deactivate/Activate updates profiles.status via updateProfileStatus.

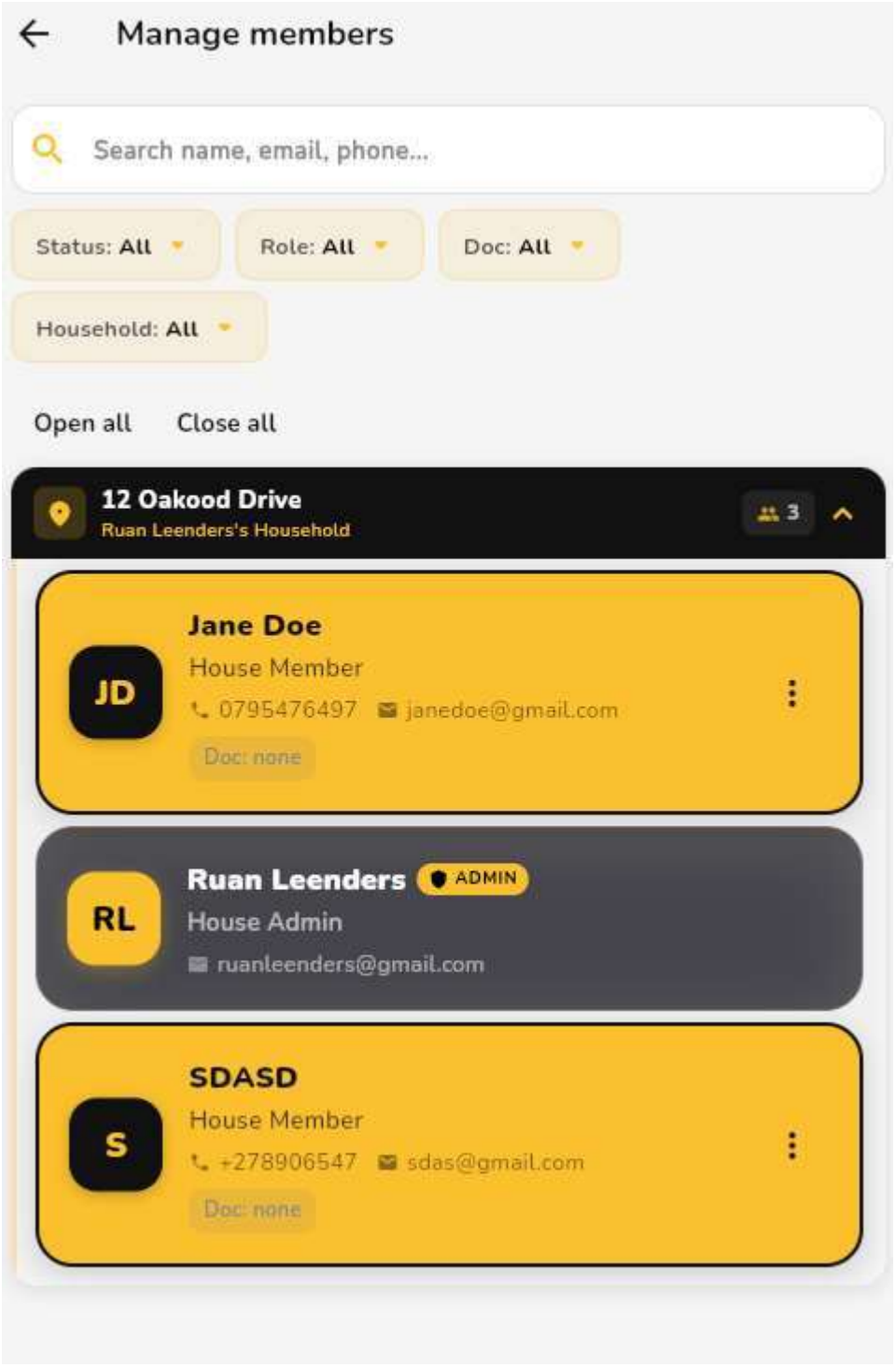
Code

- **Screen:** neighbourhood_members_screen.dart – search, filters, grouped list, expand/collapse.
- **Components:** _MemberHouseholdSection (household header + dropdown), _NeighbourhoodMemberCard (admin vs member styling), _FilterChip, _HouseholdFilterChip.
- **Service:** supabase_service.dart – getNeighbourhoodMembers() merges profiles with neighbourhood and profiles from household_members, adds household_id per member.
- **Migration:** 020_neighbourhood_members_include_household.sql – updates add_household_member and RLS for invitee profiles.

Tested / status

- **Tested:** Yes.
- **Result:** Search and filters work; members grouped by household; expand/collapse works; House Admin (dark) vs House Member (yellow) styling correct; 3-dot menu Deactivate/Activate works; invitees (e.g. SDASD, Jane Doe) appear in Manage Members.
- **Note:** Members without household_id (e.g. profiles with neighbourhood but not in any household) are excluded from the grouped list.


All vehicles
Search plate, make, color, driver, household...
Status: All
Sort: Household
Color: All
Household: All
Open all
Close all
12 Oakood Drive
Ruan Leenders's Household
3
2
5 photos
DASD
dasd dasd
Baby Blue
2025
Owner
Ruan Leenders's Household
5 photos
CA12573-92
Toyot Corolla
Dark Green
2025
Ruan Leenders
Ruan Leenders's Household


Manage members
Search name, email, phone...
Status: All
Role: All
Doc: All
Household: All
Open all
Close all
12 Oakood Drive
Ruan Leenders's Household
3
Jane Doe
House Member
0795476497
janedoe@gmail.com
Doc: none
JD
Ruan Leenders
ADMIN
House Admin
ruanleenders@gmail.com
RL
SDASD
House Member
+278906547
sdas@gmail.com
Doc: none
S


Manage members
Search name, email, phone...
Status: All
Role: All
Doc: All
Household: All
Open all
Close all
12 Oakood Drive
Ruan Leenders's Household
3
Jane Doe
House Member
0795476497
Deactivate
RL
Ruan Leenders
ADMIN
House Admin
ruanleenders@gmail.com
SDASD
House Member
+27
Doc: none

Feature 4: Vote reason dropdown

Description

When voting on an alert, users can optionally choose a reason (e.g. Known resident, Delivery driver) before tapping Friendly or Foe. The reason is stored with the vote and shown in the vote history.

Where it appears

- **Screen:** Alert Detail (opened from Alerts tab when you tap an alert).
- **Position:** "Reason (optional)" dropdown above the vote buttons (Friendly / Foe).
- **When:** Only on active alerts; dropdown is optional and can be left empty.

Database

- **Table:** votes
- **Column:** reason (TEXT, nullable)
- **Migration:** 021_vote_reason.sql adds the column.
- **Values:** known_resident, delivery, visitor, suspicious, other, or null (no reason).

How it works (flow)

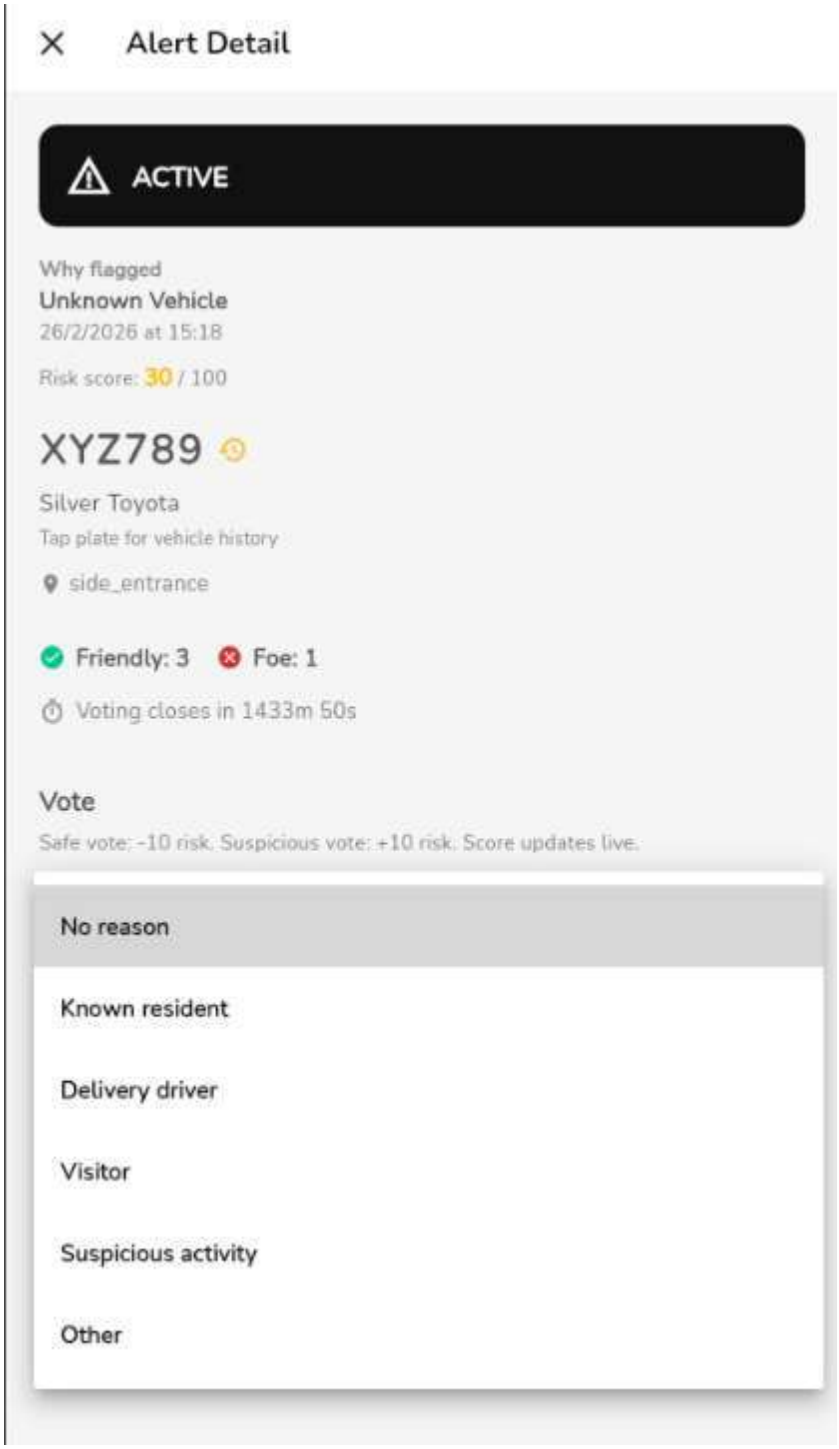
1. User opens an active alert from the Alerts tab.
2. On Alert Detail, the "Reason (optional)" dropdown lists: No reason, Known resident, Delivery driver, Visitor, Suspicious activity, Other.
3. User selects a reason (or leaves "No reason").
4. User taps Friendly or Foe.
5. App calls castVote / voteAlertSupabase with reason (or null).
6. Vote is inserted into votes with reason set.
7. SnackBar confirms the vote was submitted.

Code

- **Frontend:** alert_detail_screen.dart – StatefulWidget with _selectedReason; dropdown above vote buttons; castVote / voteAlertSupabase pass reason.
- **Service:** supabase_service.dart – voteAlertSupabase(alertId, value, reason) inserts into votes with optional reason.
- **Migration:** backend/migrations/021_vote_reason.sql – ALTER TABLE votes ADD COLUMN IF NOT EXISTS reason TEXT.

Tested / status

- **Tested:** Yes.
- **Result:** Reason dropdown appears on Alert Detail; selecting a reason and voting submits correctly; SnackBar appears; votes with reason are stored.


Alert Detail
ACTIVE
Why flagged
Unknown Vehicle
26/2/2026 at 15:18
Risk score: 30 / 100
XYZ789
Silver Toyota
Tap plate for vehicle history
side_entrance
Friendly: 3
Foe: 1
Voting closes in 1433m 50s
Vote
Safe vote: -10 risk. Suspicious vote: +10 risk. Score updates live.
No reason
Known resident
Delivery driver
Visitor
Suspicious activity
Other

Feature 5: Realtime vote counts

Description

When another user votes on an alert, the vote counts (Friendly / Foe) update live on all open Alert Detail screens without refresh.

Where it appears

- **Screen:** Alert Detail (opened from Alerts tab).
- **Position:** Vote counts shown near the Friendly / Foe buttons.
- **When:** App subscribes to Realtime on the votes table; new votes trigger a reload of alerts and vote details.

Database

- **Table:** votes
- **Realtime:** votes must be in the supabase_realtime publication.
- **Logic:** On INSERT on votes, the app reloads alerts and vote details for the affected alert.

How it works (flow)

1. User A opens Alert Detail for an active alert on Device A (or Chrome tab).
2. App subscribes to Realtime on votes via subscribeToVotes() in AppProvider.
3. User B votes on the same alert on Device B (or another tab).
4. Supabase Realtime sends an INSERT event for the new vote.
5. App receives the event and reloads alerts and vote details.
6. User A sees updated vote counts without refresh.

Code

- **Provider:** app_provider.dart – subscribeToVotes() listens to votes Realtime channel; on new vote, reloads alerts and vote details.
- **Service:** supabase_service.dart – getVotesForAlert(alertId) fetches vote counts; subscribeToVotes uses Supabase Realtime.
- **Realtime setup:** Add votes to the Supabase Realtime publication (e.g. via migration or Dashboard → Database → Replication).

Tested / status

- **Tested:** Yes.
- **Result:** Vote counts update live when another device/tab votes; no manual refresh needed.
- **Note:** Ensure votes is in the Realtime publication (Supabase Dashboard → Database → Replication, or a migration that adds votes to supabase_realtime).

Feature 6: Doc approval (admin)

Description

Neighbourhood admins can approve or reject proof-of-residence and ID documents submitted by members. Pending documents appear in the admin dashboard with links to view them. Approve sets doc_status to verified; Reject sets it to rejected.**Where it appears**

- **Entry:** Admin Dashboard → "Pending document approvals".
- **Position:** Cards for each user with pending docs; links for "Proof of residence" and "ID document"; Approve and Reject buttons.
- **When:** Only for profiles with doc_status = 'pending' and at least one of proof_of_residence_uri or id_document_uri set.

Database

- **Table:** profiles
- **Columns:** proof_of_residence_uri, id_document_uri, doc_status
- **Logic:** doc_status is 'none' | 'pending' | 'verified' | 'rejected'. Admins load profiles with doc_status = 'pending' and non-null document URIs.

How it works (flow)

1. Member uploads proof of residence and/or ID in Profile → Documents.
2. Backend stores them as data URIs (or URLs) and sets doc_status = 'pending'.
3. Admin opens Admin Dashboard and sees "Pending document approvals".
4. Admin taps "Proof of residence" or "ID document" → document opens in a new browser tab (blob URL for data URIs).
5. Admin taps Approve or Reject.
6. App calls updateProfileDocStatus(profileId, 'verified' | 'rejected').
7. Supabase updates profiles.doc_status; the card is removed from the pending list.

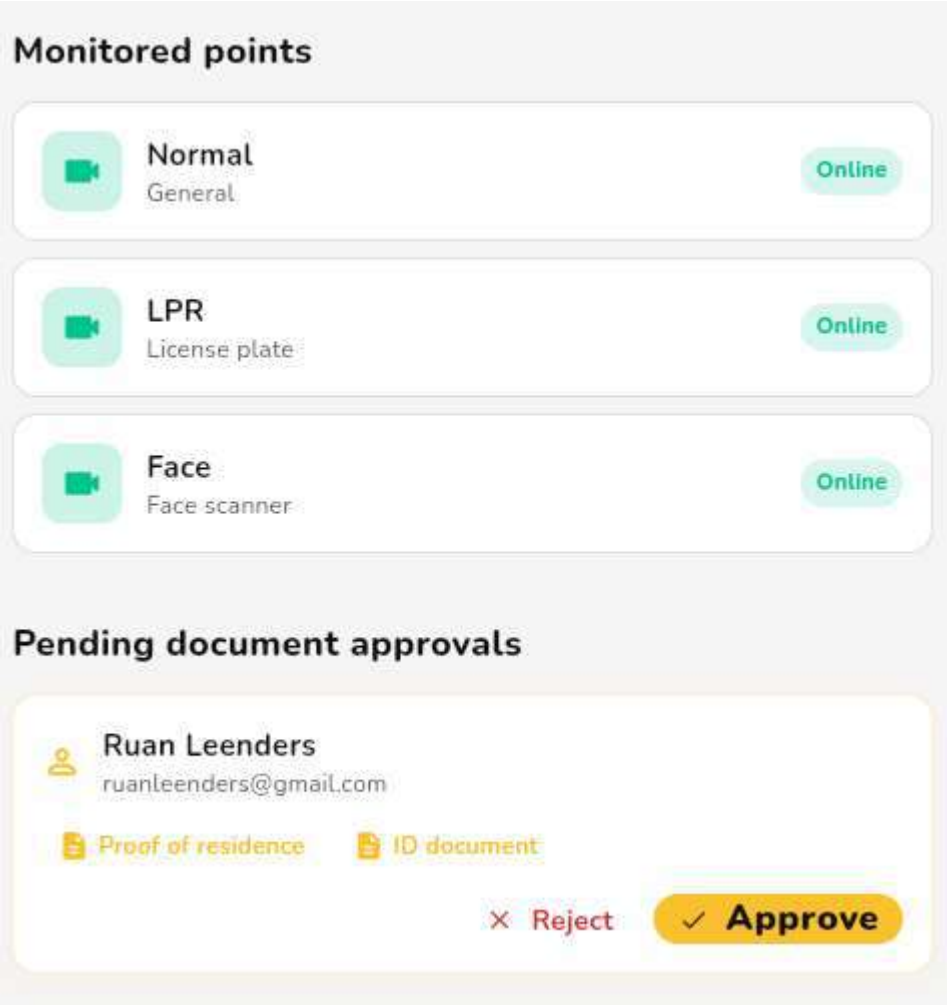
Code

- **Screen:** admin_dashboard_screen.dart – _PendingApprovalCard shows user, document links, Approve/Reject.
- **Links:** _DocLink uses launchDocInNewTab() from utils/launch_doc.dart (web: blob URL; mobile: launchUrl).
- **Service:** supabase_service.dart – getProfilesWithPendingDocApproval(), updateProfileDocStatus().
- **Provider:** app_provider.dart – loads pending approvals and triggers refresh after approve/reject.

Tested / status

- **Tested:** Yes.

- **Result:** Pending docs appear; document links open in a new tab; Approve/Reject update status and remove the card.
- **Note:** Documents are stored as data URIs; web uses blob URLs to avoid long-URL limits when opening in a new tab.



Feature 7: Admin face approval – test steps

Description

Neighbourhood admins can approve or reject face biometric enrollments. Profiles with face_bio_status = 'uploaded' appear in the admin dashboard for review.

Where to test

- **Device:** Android emulator or physical device (e.g. Huawei); web usually does not support camera.
- **Entry:** Admin Dashboard → Face enrollment approvals.

Test steps

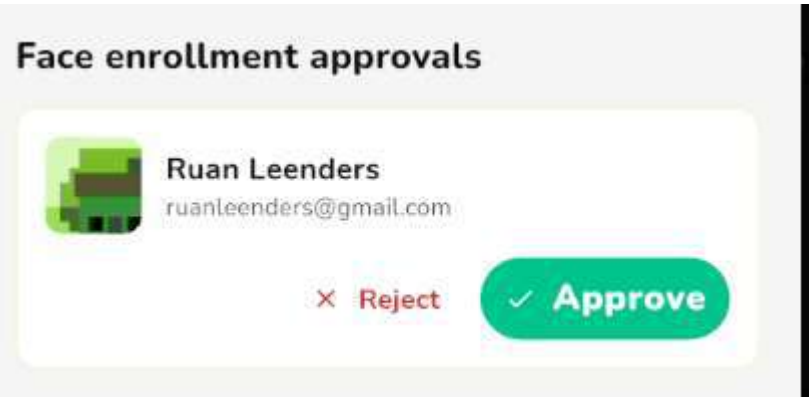
1. Log in as a **normal user**.
2. Go to **Profile** → **Facial Biometrics** → capture face.
3. Log out.
4. Log in as **neighbourhood admin**.
5. Open **Neighbourhood Admin** (Profile or Household).
6. Scroll to **Face enrollment approvals**.
7. Use **Approve** or **Reject** for the pending face.

Database

- **Table:** profiles
- **Columns:** face_image_uri, face_bio_status
- **Logic:** face_bio_status = 'uploaded' → pending approval; admin sets to 'verified' or 'rejected'.

Tested / status

- **Tested:** Yes.
- **Note:** Requires Android device or emulator with camera.



Feature 8: Face quality checks – test steps

Description

Face capture enforces quality rules (min size, min dimensions). Poor captures show a SnackBar; good captures go to the review screen.

Where to test

- **Device:** Android emulator or physical device; web usually does not support camera.
- **Entry:** Profile → Facial Biometrics → capture face.

Test steps

1. Go to **Profile** → **Facial Biometrics** → capture face.
2. **Quality fail:** Stay far from the camera → SnackBar: "Face too small. Move closer to the camera."
3. **Quality pass:** Move closer, capture normally → Review screen → Confirm → saved.

Code

- **Screen:** face_scanner_screen.dart – _checkQuality() enforces min file size (15KB) and min dimensions (200×200 px).

Tested / status

- **Note:** Requires Android device or emulator with camera.

Feature 9: Push notifications – test steps

Description

When an alert is created, the app receives a push notification (e.g. "Plate X at Entry Point"). Tapping it opens the Alert Detail screen.

Where to test

- **Device:** Android emulator or physical device; not supported on web.
- **Entry:** App running in background; create an alert via Supabase SQL or backend webhook.

Requirements

- Firebase Cloud Messaging (FCM) configured for the app.
- google-services.json in the Android project.
- App built in release or debug with FCM.

Test steps

1. Run the app on your Huawei (or Android device).
2. Create an alert (Supabase SQL or backend webhook).
3. A push notification should appear: "Plate X at Entry Point".

4. Tap the notification → Alert Detail opens.

If push notifications don't appear

- Confirm FCM is set up in the project.
- Check that the device has Google Play Services (most Huawei do).
- Ensure notifications are allowed for the app in Settings.
- For Huawei with HMS instead of GMS, FCM may not work; HMS would need separate setup.

Code

- **Backend:** notifications.py – dispatch_notification() sends push via FCM.
- **Frontend:** FCM registration and notification handling in the app.

Tested / status

- **Tested:** Yes.
- **Note:** Requires FCM setup and an Android device with Google Play Services (or HMS if using Huawei push).

+10 per Suspicious (Foe) vote	Yes
-10 per Safe (Friendly) vote	Yes
Voting window: 15 minutes	Yes (`webhooks.py`)
One vote per user	Yes (`UNIQUE (alert_id, voter_id)`)
Cannot vote after window closes	Yes (trigger + API check)
Formula: `base + (suspicious*10) - (safe*10)`	Yes
Score clamped 0–100	Yes
`POST /alerts/{id}/vote`	Yes
Auto recalculation of score	Yes (DB trigger)
Alert closes after timer expires	Yes (trigger sets status)

2. WORKPLACE JOURNEY AND REFLECTIONS

2.1 Significant experiences and challenges

The main project was a neighbourhood security platform (Neighbourhood Bot) with a Flutter app, FastAPI backend, and Supabase. One of the first big challenges was understanding how everything fits together: auth, RLS, and how neighbourhood admins see and approve user data.

A concrete example was the approval flow. As a neighbourhood admin, I could approve docs and face enrollment, but the approval cards showed "—" instead of household details. After digging into the code and DB, I found that the admin's neighbourhood was used to filter households, while the approval list could include users from other neighbourhoods. That mismatch meant household info was never returned. The fix was to treat neighbourhood admins as platform-level admins who can see all neighbourhoods, and to add RLS policies so they can read households and members across neighbourhoods.

Another issue was the face photo disappearing after verification. The DB had the correct face_image_uri and face_bio_status, but the profile screen still showed "Not enrolled". The problem was in the frontend: loadProfileFromSupabase wasn't mapping those fields into the profile model. Once we added faceImageUri and faceBioStatus to the profile loading logic, the face photo and status showed correctly.

2.2 Lessons learned and skills gained

Technical

- How RLS works in Supabase and how to design policies for different roles.
- How to trace data from DB → backend → frontend and find where mappings or filters break.
- Using migrations to evolve schema and policies safely.
- Debugging by inspecting the DB and comparing it with what the UI shows.

Process

- The importance of checking the DB first when something looks wrong in the UI.
- Breaking problems into smaller steps (e.g. RLS, data mapping, UI display).
- Using inspection scripts to understand data and access patterns.

Growth

- Getting more comfortable with full-stack debugging across Flutter, FastAPI, and Supabase.
- Learning to question assumptions (e.g. "one admin per neighbourhood") and adjust the design when it doesn't match real use.
- Paying more attention to how data flows through the system and where it can get lost or filtered out.

3. MENTOR FEEDBACK, WORK ETHICS, AND SOFT SKILLS

Mentor feedback stressed the importance of inspecting the database before changing code. When something looked wrong in the UI, the first step was to query the database and confirm what was actually stored.

That habit helped avoid assumptions and led to fixes that addressed the real cause instead of surface-level symptoms.

Another recurring theme was discussing design choices before implementing them. For example, when we realised neighbourhood admins could not see household details for users in other neighbourhoods, we explored different models (super-admin vs per-neighbourhood admin) and agreed on the super-admin approach before writing any code. This helped us align on the right solution and avoid unnecessary rework.

In terms of work ethics and attitudes, persistence was a big factor.

When household info or the face photo disappeared, the natural reaction was to keep digging until the root cause was clear. That meant tracing data from the database through the backend and into the frontend, rather than guessing or making quick fixes.

Clarity was also important. We tried to explain technical problems and solutions in plain language, even when the underlying systems were complex. That made it easier to collaborate and to understand what was going wrong.

Iteration was another guiding principle. We addressed one issue at a time—household display, address formatting, face photo display, fullscreen view—instead of trying to fix everything at once. That kept changes manageable and made it easier to see what worked and what did not.

Soft skills that came up included problem-solving and communication. Breaking vague issues like “it’s not showing right” into specific, testable hypotheses made debugging more effective. Describing technical problems and solutions in a way others could follow helped keep the work aligned and understandable.

Attention to detail mattered too. Small things like address formatting or missing profile fields had a noticeable impact on user experience. Being patient enough to inspect, read, and understand before making changes reduced the risk of introducing new bugs or oversimplifying the problem.

Overall, the project showed how important it is to understand the full system, verify assumptions with data, and iterate in small steps while keeping the user’s experience in mind.

BELGIUM CAMPUS iTversity
It's the way we're wired

3 MONTH MENTOR REPORT

Student Name: Ruan Leenders
Report Date: 2 December 2025

HOW WOULD YOU RATE THE STUDENT'S PERFORMANCE SO FAR?

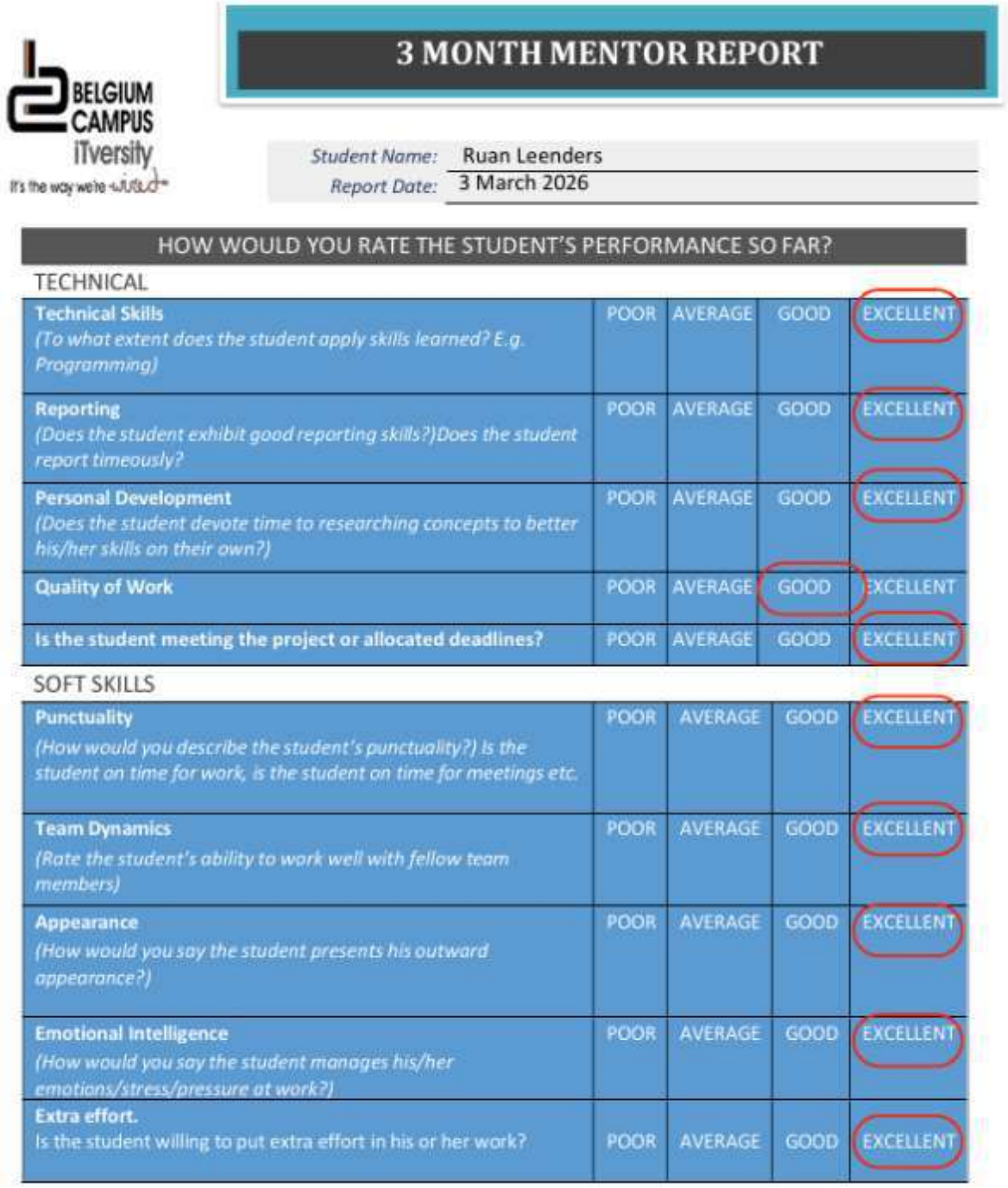
TECHNICAL

Technical Skills *(To what extent does the student apply skills learned? E.g. Programming)*	POOR	AVERAGE	GOOD	EXCELLENT
Reporting *(Does the student exhibit good reporting skills?)Does the student report timeously?*	POOR	AVERAGE	GOOD	EXCELLENT
Personal Development *(Does the student devote time to researching concepts to better his/her skills on their own?)*	POOR	AVERAGE	GOOD	EXCELLENT
Quality of Work	POOR	AVERAGE	GOOD	EXCELLENT
Is the student meeting the project or allocated deadlines?	POOR	AVERAGE	GOOD	EXCELLENT

SOFT SKILLS

Punctuality *(How would you describe the student's punctuality?) Is the student on time for work, is the student on time for meetings etc.*	POOR	AVERAGE	GOOD	EXCELLENT
Team Dynamics *(Rate the student's ability to work well with fellow team members)*	POOR	AVERAGE	GOOD	EXCELLENT
Appearance *(How would you say the student presents his outward appearance?)*	POOR	AVERAGE	GOOD	EXCELLENT
Emotional Intelligence *(How would you say the student manages his/her emotions/stress/pressure at work?)*	POOR	AVERAGE	GOOD	EXCELLENT
Extra effort. Is the student willing to put extra effort in his or her work?	POOR	AVERAGE	GOOD	EXCELLENT

Figure 1-Mentor Review 2 December 2025 (3 months)



BELGIUM CAMPUS iTversity
It's the way we're wired

3 MONTH MENTOR REPORT

Student Name: Ruan Leenders
Report Date: 3 March 2026

HOW WOULD YOU RATE THE STUDENT'S PERFORMANCE SO FAR?

TECHNICAL

Technical Skills *(To what extent does the student apply skills learned? E.g. Programming)*	POOR	AVERAGE	GOOD	EXCELLENT
Reporting *(Does the student exhibit good reporting skills?)Does the student report timeously?*	POOR	AVERAGE	GOOD	EXCELLENT
Personal Development *(Does the student devote time to researching concepts to better his/her skills on their own?)*	POOR	AVERAGE	GOOD	EXCELLENT
Quality of Work	POOR	AVERAGE	GOOD	EXCELLENT
Is the student meeting the project or allocated deadlines?	POOR	AVERAGE	GOOD	EXCELLENT

SOFT SKILLS

Punctuality *(How would you describe the student's punctuality?) Is the student on time for work, is the student on time for meetings etc.*	POOR	AVERAGE	GOOD	EXCELLENT
Team Dynamics *(Rate the student's ability to work well with fellow team members)*	POOR	AVERAGE	GOOD	EXCELLENT
Appearance *(How would you say the student presents his outward appearance?)*	POOR	AVERAGE	GOOD	EXCELLENT
Emotional Intelligence *(How would you say the student manages his/her emotions/stress/pressure at work?)*	POOR	AVERAGE	GOOD	EXCELLENT
Extra effort. Is the student willing to put extra effort in his or her work?	POOR	AVERAGE	GOOD	EXCELLENT

Figure 2-Mentor Review 3 March 2026 (3 months)

4. CONCLUSION

Over the past six months, from September 2025 to March 2026, my journey at Delegate IT has been a period of profound technical and professional growth. Working on diverse, high-impact platforms—including the MAN Total Cost of Ownership (TCO) tool, the Special Price Request (SPR) system, the MAN Contract Management System, and the Neighbourhood Bot security app —has allowed me to transition from executing isolated bug fixes to architecting full-scale, production-ready features.

Whether I was optimizing database queries to reduce load times from seconds to milliseconds , implementing complex multi-approver workflows , or building a digital signature engine from scratch , these projects have significantly sharpened my full-stack development capabilities across React, TypeScript, C#, Flutter, and PostgreSQL.Beyond writing code, the most valuable takeaways from this experience stem from the guidance of my mentors and the practical realities of software engineering. I learned the critical importance of a "database-first" debugging approach—always verifying the data at its source before making assumptions in the user interface.

By tackling intricate challenges like Row Level Security (RLS) mismatches, real-time state synchronization, and complex data mapping, I developed a more resilient, iterative approach to problem-solving. I also realized that writing maintainable software goes hand-in-hand with clear communication; discussing design choices before implementation and breaking down vague issues into testable hypotheses proved essential to delivering reliable solutions.

Ultimately, this portfolio represents more than just a collection of completed tasks; it reflects my evolution into a more thoughtful, detail-oriented, and capable software developer. I am walking away with a deep understanding of how to build secure, user-centric systems that solve real-world problems. I am incredibly grateful for the mentorship, the collaborative environment, and the trust placed in me to build these systems. I am excited to carry these technical skills, strong work ethics, and analytical strategies forward into the next chapter of my software engineering career.

During this portfolio of evidence, I presented the main work and contributions I made while working on several systems and projects. These included improvements to the MAN TCO system, workflow features, deal management tools, and parts of the neighborhood security application. Through these tasks I was able to strengthen my problem-solving abilities, improve existing systems, and contribute practical features that make the applications easier to use and more efficient. Working on real systems gave me valuable insight into how software is developed, maintained, and improved in a professional environment.

Overall, the experience helped me grow both technically and professionally, as I learned how to approach problems logically, work with existing codebases, and deliver solutions that meet business needs. The AIT program played an important role in preparing me for this work by giving me the foundational knowledge and practical skills needed to handle real development tasks. In my opinion, the program provides a strong base for students entering the industry, and continuing to focus on practical, real-world projects and collaboration with companies would make it even more valuable for future students.