RIVET Pro Multi-Agent Pipeline System PRD

Bottom Line Up Front

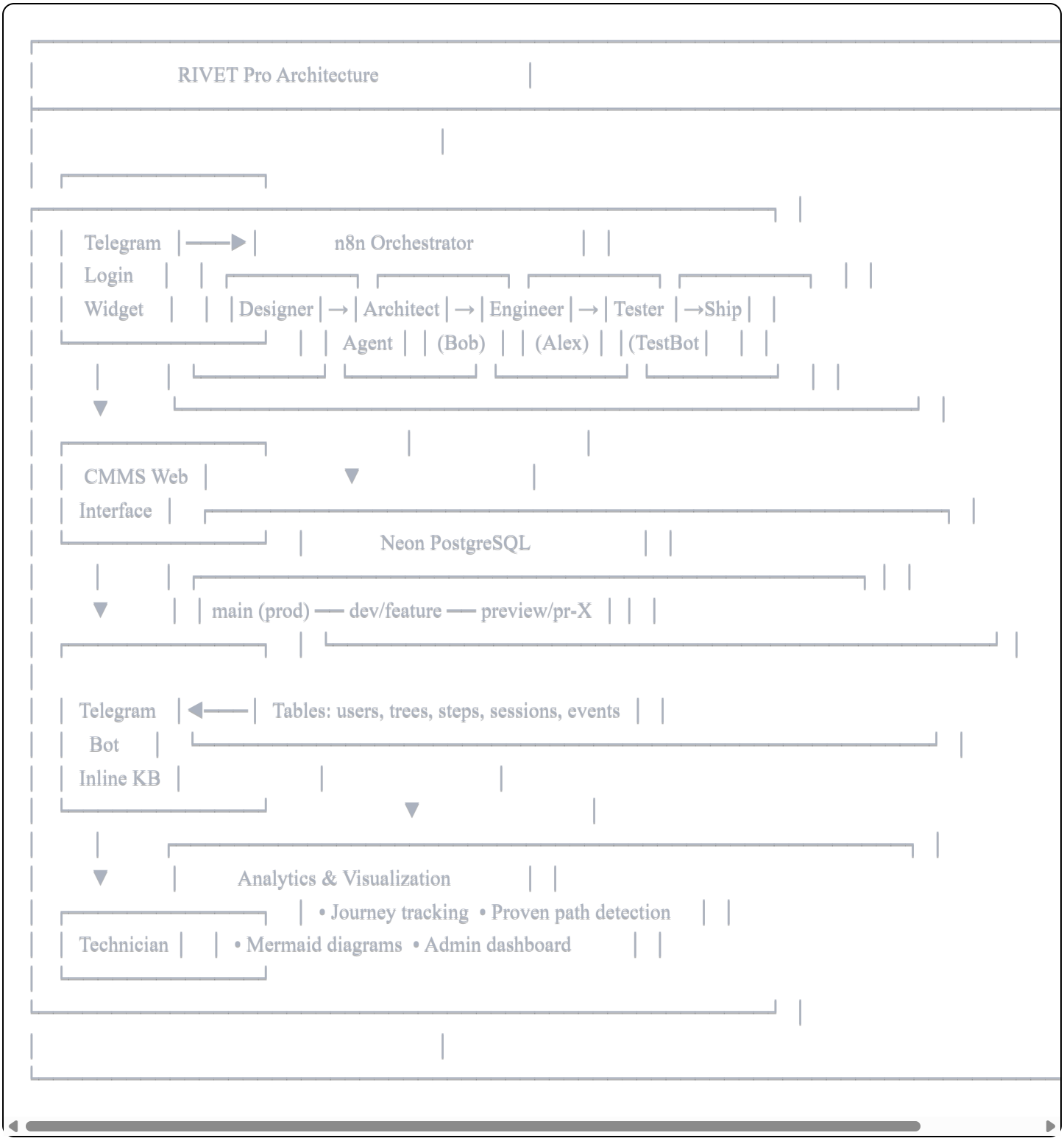
RIVET Pro is a multi-agent pipeline system for CMMS (Computerized Maintenance Management System) troubleshooting, combining n8n workflow orchestration with Telegram-based user interaction. The system enables maintenance technicians to receive interactive, step-by-step troubleshooting guidance through Telegram inline buttons, while tracking every interaction to identify "proven paths" that consistently resolve issues. Five specialized AI agents (Designer → Architect → Engineer → Tester → Deployer) collaborate through n8n workflows to generate and refine troubleshooting content, with Neon PostgreSQL providing database branching for isolated dev/test/prod environments. This PRD provides complete implementation specifications—database schemas, API patterns, workflow architectures, and acceptance criteria—ready for the Ralph autonomous agent system to build.

---

1. System architecture overview

Core components

RIVET Pro consists of five interconnected subsystems: **Authentication** (Telegram Login Widget for web, Telegram Bot for mobile), **Content Pipeline** (5-agent n8n workflow chain), **Troubleshooting Delivery** (Telegram inline buttons), **Analytics Engine** (journey tracking and proven path identification), and **Admin Dashboard** (observability and content management).



---

2. User authentication and identity

Telegram Login Widget implementation

The system uses Telegram as the primary identity provider, leveraging the **Telegram Login Widget** for web authentication and the **Telegram Bot API** for mobile users.

Widget HTML Implementation:

```html
<script async src="https://telegram.org/js/telegram-widget.js?22"
  data-telegram-login="RivetProBot"
  data-size="large"
  data-auth-url="https://rivetpro.app/auth/telegram/callback"
  data-request-access="write">
</script>
```

Backend Hash Verification (Node.js):

```javascript
const crypto = require('crypto');

function verifyTelegramAuth(data, botToken) {
  const { hash, ...userData } = data;

  // Create secret key from bot token
  const secretKey = crypto.createHash('sha256')
    .update(botToken)
    .digest();

  // Build data check string (sorted alphabetically)
  const dataCheckString = Object.keys(userData)
    .sort()
    .filter(key => userData[key] !== undefined && userData[key] !== '')
    .map(key => `${key}=${userData[key]}`)
    .join('\n');

  // Calculate HMAC
  const hmac = crypto.createHmac('sha256', secretKey)
    .update(dataCheckString)
    .digest('hex');

  return hmac === hash;
}
```

User data schema

Telegram Login returns these fields, with `id` (Telegram User ID) serving as the **universal user key** across all system components:

Field	Type	Required	Description
`id`	number	✓	Unique Telegram user ID (permanent, universal key)
`first_name`	string	✓	User's first name
`last_name`	string		User's last name
`username`	string		Telegram @username
`photo_url`	string		Profile photo URL
`auth_date`	number	✓	Unix timestamp of authentication
`hash`	string	✓	HMAC-SHA-256 verification hash

Session management pattern

```javascript
```

```
// POST /auth/telegram/callback
app.post('/auth/telegram/callback', async (req, res) => {
  const data = req.body;

  // 1. Verify hash
  if (!verifyTelegramAuth(data, process.env.BOT_TOKEN)) {
    return res.status(401).json({ error: 'Invalid authentication' });
  }

  // 2. Check freshness (max 24 hours)
  if (Date.now()/1000 - data.auth_date > 86400) {
    return res.status(401).json({ error: 'Authentication expired' });
  }

  // 3. Find or create user
  const user = await db.query(`
    INSERT INTO users (telegram_id, telegram_first_name, telegram_last_name,
                       telegram_username, telegram_photo_url, telegram_auth_date)
    VALUES ($1, $2, $3, $4, $5, to_timestamp($6))
    ON CONFLICT (telegram_id) DO UPDATE SET
      telegram_first_name = EXCLUDED.telegram_first_name,
      telegram_username = EXCLUDED.telegram_username,
      telegram_photo_url = EXCLUDED.telegram_photo_url,
      last_seen_at = NOW()
    RETURNING *
  `, [data.id, data.first_name, data.last_name, data.username, data.photo_url, data.auth_date]);

  // 4. Create JWT session
  const token = jwt.sign({ telegramId: data.id }, process.env.JWT_SECRET, { expiresIn: '7d' });

  res.cookie('session', token, { httpOnly: true, secure: true, sameSite: 'lax' });
  res.redirect('/dashboard');
});
```

Acceptance criteria for authentication

- [ ] **AC-AUTH-1**: Telegram Login Widget renders on CMMS web login page
- [ ] **AC-AUTH-2**: Backend verifies HMAC-SHA-256 hash correctly
- [ ] **AC-AUTH-3**: Authentication data older than 24 hours is rejected
- [ ] **AC-AUTH-4**: User record created/updated in `users` table on first/subsequent logins
- [ ] **AC-AUTH-5**: JWT session token issued with 7-day expiry
- [ ] **AC-AUTH-6**: Telegram user ID used consistently as foreign key across all tables

3. Multi-agent pipeline architecture

n8n workflow orchestration

The content generation pipeline uses five specialized agents, each implemented as a separate n8n workflow. A **Parent Orchestrator** workflow coordinates sequential execution with human approval gates.

Pipeline Flow:

```
[Webhook Trigger: /pipeline/start]
  ↓
[Postgres: Create pipeline record]
  ↓
[Execute Sub-workflow: Designer Agent] → Generates Mermaid diagram
  ↓
[Telegram: Send diagram for approval] ← Human gate
  ↓
[Wait Node: Design Approval]
  ↓
[Execute Sub-workflow: Architect Agent (Bob)] → System design
  ↓
[Execute Sub-workflow: Engineer Agent (Alex)] → Implementation
  ↓
[Execute Sub-workflow: Tester Agent (TestBot)] → Validation
  ↓
[Wait Node: Deployment Approval] ← Human gate
  ↓
[Execute Sub-workflow: Deployer Agent (Ship)] → Production release
  ↓
[Postgres: Update pipeline status = 'completed']
```

Agent workflow structure

Each agent workflow follows this template:

```
javascript
```

```
// Execute Sub-workflow Trigger (start node)
{
  "inputDataMode": "Define using fields below",
  "fields": [
    { "name": "pipelineId", "type": "string" },
    { "name": "stage", "type": "string" },
    { "name": "inputData", "type": "object" },
    { "name": "previousOutput", "type": "object" }
  ]
}

// Workflow sequence:
// 1. Update pipeline_stages status = 'running'
// 2. Prepare AI prompt with context
// 3. Call Claude API via AI Agent node
// 4. Validate output
// 5. Update pipeline_stages status = 'completed'
// 6. Return structured output to parent
```

Designer Agent specifics

The Designer Agent generates Mermaid diagrams for troubleshooting decision trees:

javascript

```
// Claude prompt for Designer Agent
const designerPrompt = `
You are the Designer Agent for RIVET Pro. Generate a Mermaid flowchart
for a troubleshooting decision tree.

REQUIREMENTS:
- Use flowchart TD (top-down) direction
- Diamond shapes {text} for questions/decisions
- Rectangle shapes [text] for actions
- Stadium shapes ([text]) for terminal states (resolution/escalation)
- Label edges with user response options (Yes/No, specific choices)
- Include safety warnings where applicable
- Maximum 15 steps per tree

INPUT: ${JSON.stringify(inputData)}

OUTPUT FORMAT:
\`\`\`mermaid
flowchart TD
   START[Select Issue]
   Q1{Is power LED on?}
   ...
\`\`\`
`;
```

Human approval gate implementation

javascript

```
// Telegram approval with Wait node
{
  "operation": "sendAndWait",
  "chatId": "{{ $env.APPROVAL_CHAT_ID }}",
  "message": "🔔 **Design Review Required**\n\nPipeline: {{ $json.pipelineId }}\nStage: Designer Agent\n\n[View Diagra
  "approvalOptions": {
    "approvalType": "double",
    "approveLabel": "✅ Approve Design",
    "declineLabel": "❌ Request Changes"
  },
  "limitWaitTime": true,
  "limitAmount": 24,
  "limitUnit": "hours"
}
```

State tracking and backlog.md generation

Pipeline state is tracked in the `pipeline_stories` and `agent_outputs` tables. A scheduled workflow generates `backlog.md` for observability:

```
javascript
```

```
// Generate backlog.md content
async function generateBacklog() {
  const pipelines = await db.query(`
    SELECT ps.*,
           json_agg(ao.* ORDER BY ao.execution_order) as agent_outputs
    FROM pipeline_stories ps
    LEFT JOIN agent_outputs ao ON ps.id = ao.pipeline_id
    WHERE ps.status IN ('pending', 'running')
    GROUP BY ps.id
    ORDER BY ps.created_at DESC
  `);

  let markdown = `# RIVET Pro Pipeline Backlog\n\n`;
  markdown += `*Generated: ${new Date().toISOString()}*\n\n`;

  for (const pipeline of pipelines) {
    markdown += `## Pipeline: ${pipeline.pipeline_name}\n`;
    markdown += `- **Status**: ${pipeline.status}\n`;
    markdown += `- **Current Agent**: ${pipeline.current_agent || 'N/A'}\n`;
    markdown += `- **Started**: ${pipeline.started_at}\n\n`;

    markdown += `### Agent Progress\n`;
    for (const agent of pipeline.agent_outputs) {
      const status = agent.status === 'success' ? '✅' : agent.status === 'running' ? '🔄' : '⏳';
      markdown += `${status} ${agent.agent_name}: ${agent.status}\n`;
    }
    markdown += `\n---\n\n`;
  }

  return markdown;
}
```

Acceptance criteria for pipeline

- [ ] **AC-PIPE-1**: Parent orchestrator triggers via webhook POST /pipeline/start
- [ ] **AC-PIPE-2**: Each agent workflow callable via Execute Sub-workflow node
- [ ] **AC-PIPE-3**: Pipeline state persisted to pipeline_stories table after each stage
- [ ] **AC-PIPE-4**: Telegram notification sent for Design approval with approve/decline buttons
- [ ] **AC-PIPE-5**: Wait node pauses execution until human response (max 24h timeout)
- [ ] **AC-PIPE-6**: backlog.md generated every 5 minutes via scheduled workflow
- [ ] **AC-PIPE-7**: Failed stages trigger error workflow with Telegram notification
- [ ] **AC-PIPE-8**: Retry logic with exponential backoff (up to 5 attempts)

4. Interactive troubleshooting system

Telegram inline button interface

Troubleshooting is delivered as a tap-through experience using Telegram's **inline keyboard buttons**. Each step displays content, optional media, and navigation buttons.

Message Structure:

javascript

```
async function sendTroubleshootingStep(bot, chatId, step) {
  const keyboard = step.buttons.map(btn => [{
    text: btn.label,
    callback_data: `step_${step.id}_${btn.id}`  // Max 64 bytes UTF-8
  }]);

  // Add navigation row
  keyboard.push([
    { text: '⬅️ Back', callback_data: `nav_back_${step.id}` },
    { text: '🔄 Restart', callback_data: 'nav_restart' }
  ]);

  const message = formatStepMessage(step);

  if (step.images && step.images.length > 0) {
    // Send photo with inline keyboard
    await bot.sendPhoto(chatId, step.images[0].url, {
      caption: message,
      parse_mode: 'HTML',
      reply_markup: { inline_keyboard: keyboard }
    });
  } else {
    // Send text message with inline keyboard
    await bot.sendMessage(chatId, message, {
      parse_mode: 'HTML',
      reply_markup: { inline_keyboard: keyboard }
    });
  }
}

function formatStepMessage(step) {
  let message = `<b>${step.title}</b>\n\n`;
  message += step.content + '\n\n';

  // Safety warning formatting
  if (step.metadata?.safety_warning) {
    message += `<blockquote>⚠️ <b>SAFETY WARNING</b>\n${step.metadata.safety_warning}</blockquote>\n\n`;
  }

  // Tool requirements
  if (step.metadata?.tools_required) {
    message += `🔧 <b>Tools needed:</b> ${step.metadata.tools_required.join(', ')}\n\n`;
  }
```

```
  return message;
}
```

Callback query handling

javascript

```
bot.on('callback_query', async (query) => {
  const chatId = query.message.chat.id;
  const userId = query.from.id;
  const data = query.data;

  // CRITICAL: Always answer to dismiss loading indicator
  await bot.answerCallbackQuery(query.id);

  // Parse callback data
  if (data.startsWith('step_')) {
    const [_, stepId, buttonId] = data.split('_');
    await handleStepSelection(chatId, userId, stepId, buttonId, query.message.message_id);
  } else if (data.startsWith('nav_back')) {
    await handleBackNavigation(chatId, userId, query.message.message_id);
  } else if (data === 'nav_restart') {
    await handleRestart(chatId, userId, query.message.message_id);
  }
});

async function handleStepSelection(chatId, userId, currentStepId, buttonId, messageId) {
  // 1. Get current session
  const session = await getActiveSession(userId);

  // 2. Find next step based on button selection
  const currentStep = await db.query(
    'SELECT buttons FROM troubleshooting_steps WHERE id = $1',
    [currentStepId]
  );
  const button = currentStep.buttons.find(b => b.id === buttonId);
  const nextStepId = button.next_step_id;

  // 3. Log journey event
  await logJourneyEvent(session.id, userId, {
    event_type: 'button_clicked',
    step_id: currentStepId,
    button_id: buttonId,
    button_label: button.label,
    to_step_id: nextStepId
  });

  // 4. Load and display next step
  const nextStep = await db.query(
    'SELECT * FROM troubleshooting_steps WHERE id = $1',
```

```
    [nextStepId]
  );

  // 5. Edit existing message (reduces chat clutter)
  await editTroubleshootingMessage(chatId, messageId, nextStep);
}
```

Mermaid to database conversion

Troubleshooting trees are authored as Mermaid diagrams and converted to database records:

javascript

```
function parseMermaidFlowchart(mermaidCode) {
  const nodes = new Map();
  const edges = [];

  const shapePatterns = {
    'rectangle': /(\w+)\[([^\]]+)\]/g,      // A[Label]
    'diamond': /(\w+)\{([^\}]+)\}/g,        // A{Label}
    'stadium': /(\w+)\(\[([^\]]+)\]\)/g,    // A([Label])
  };

  const edgePattern = /(\w+)\s*(-->|---)\|?([^\|]*)\|?\s*(\w+)/g;

  const lines = mermaidCode.split('\n');

  for (const line of lines) {
    // Extract nodes
    for (const [shape, pattern] of Object.entries(shapePatterns)) {
      let match;
      while ((match = pattern.exec(line)) !== null) {
        const [_, nodeId, label] = match;
        if (!nodes.has(nodeId)) {
          nodes.set(nodeId, {
            id: nodeId,
            label: label.replace(/"/g, ''),
            type: shape === 'diamond' ? 'question' :
                  shape === 'stadium' ? 'solution' : 'action'
          });
        }
      }
    }

    // Extract edges
    let edgeMatch;
    while ((edgeMatch = edgePattern.exec(line)) !== null) {
      const [_, source, __, edgeLabel, target] = edgeMatch;
      edges.push({
        source,
        target,
        label: edgeLabel.trim() || null
      });
    }
  }
```

```
  return { nodes: Array.from(nodes.values()), edges };
}

async function saveTroubleshootingTree(treeId, mermaidCode) {
  const { nodes, edges } = parseMermaidFlowchart(mermaidCode);

  // Insert steps
  for (const node of nodes) {
    await db.query(`
      INSERT INTO troubleshooting_steps
        (id, tree_id, title, step_type, is_terminal)
      VALUES ($1, $2, $3, $4, $5)
    `, [node.id, treeId, node.label, node.type, node.type === 'solution']);
  }

  // Build button relationships
  for (const edge of edges) {
    await db.query(`
      UPDATE troubleshooting_steps
      SET buttons = buttons || $1::jsonb
      WHERE id = $2
    `, [
      JSON.stringify([{ id: edge.target, label: edge.label, next_step_id: edge.target }]),
      edge.source
    ]);
  }
}
```

Claude fallback for unknown issues

When no matching troubleshooting tree exists, Claude generates guidance:

```
javascript
```

```
async function handleUnknownIssue(chatId, userId, equipmentType, faultDescription) {
  // Generate troubleshooting with Claude
  const response = await anthropic.messages.create({
    model: 'claude-3-5-sonnet-20241022',
    max_tokens: 2048,
    messages: [{
      role: 'user',
      content: `Generate step-by-step troubleshooting for:
        Equipment: ${equipmentType}
        Issue: ${faultDescription}

        Format as numbered steps. Include safety warnings where applicable.
        End with escalation contact if unresolved.`
    }]
  });

  const guide = response.content[0].text;

  // Send with "Save this guide" option
  await bot.sendMessage(chatId, guide, {
    parse_mode: 'HTML',
    reply_markup: {
      inline_keyboard: [[
        { text: '💾 Save this guide', callback_data: `save_guide_${Date.now()}` },
        { text: '✅ Issue resolved', callback_data: 'resolved' }
      ]]
    }
  });

  // Store for potential saving
  await redis.setex(`pending_guide:${userId}`, 3600, JSON.stringify({
    equipmentType,
    faultDescription,
    guide
  }));
}
```

Acceptance criteria for troubleshooting

- [ ] **AC-TS-1**: Mermaid diagrams parse correctly to nodes/edges
- [ ] **AC-TS-2**: Inline keyboard displays with max 8 buttons per row
- [ ] **AC-TS-3**: callback_data stays within 64-byte limit

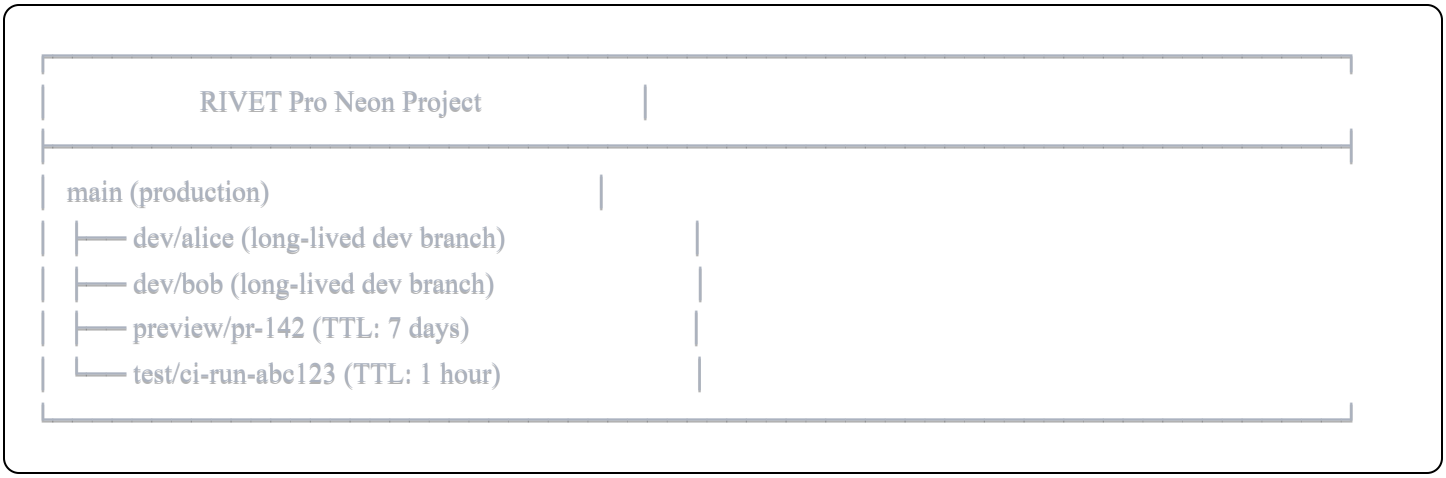
- [ ] **AC-TS-4**: Messages edit in place (no new messages per step)
- [ ] **AC-TS-5**: Images/media display with captions
- [ ] **AC-TS-6**: Safety warnings render in blockquote format
- [ ] **AC-TS-7**: Back navigation returns to previous step
- [ ] **AC-TS-8**: Claude fallback triggers for unknown equipment/faults
- [ ] **AC-TS-9**: "Save this guide" creates new troubleshooting tree draft

---

5. Neon database architecture

Database branching strategy

Neon's **copy-on-write branching** provides instant, isolated environments: neon



Branch Management API:

```javascript
```

```
// Create preview branch for PR
async function createPreviewBranch(prNumber) {
  const response = await fetch(
    `https://console.neon.tech/api/v2/projects/${PROJECT_ID}/branches`,
    {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${NEON_API_KEY}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        branch: {
          name: `preview/pr-${prNumber}`,
          parent_id: 'br-main-123456',
          expires_at: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000).toISOString()
        },
        endpoints: [{ type: 'read_write' }]
      })
    }
  );
  return response.json();
}
```

Connection strings:

```
bash

# Production (pooled for application traffic)
DATABASE_URL=postgres://user:pass@ep-xxx-pooler.us-east-2.aws.neon.tech/rivetpro

# Development (direct for migrations)
DATABASE_URL_DIRECT=postgres://user:pass@ep-xxx.us-east-2.aws.neon.tech/rivetpro
```

Complete database schema

```
sql
```

```sql
-- Enable extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pgcrypto";

-- ============================================
-- ORGANIZATIONS & USERS
-- ============================================

CREATE TABLE organizations (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(255) NOT NULL,
    slug VARCHAR(100) UNIQUE NOT NULL,
    settings JSONB DEFAULT '{}',
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    telegram_id BIGINT UNIQUE NOT NULL,  -- Universal user key
    telegram_username VARCHAR(255),
    telegram_first_name VARCHAR(255),
    telegram_last_name VARCHAR(255),
    telegram_photo_url TEXT,
    telegram_auth_date TIMESTAMPTZ,
    organization_id UUID REFERENCES organizations(id) ON DELETE SET NULL,
    role VARCHAR(50) DEFAULT 'user' CHECK (role IN ('admin', 'editor', 'user')),
    is_active BOOLEAN DEFAULT true,
    last_seen_at TIMESTAMPTZ,
    metadata JSONB DEFAULT '{}',
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_users_telegram_id ON users(telegram_id);
CREATE INDEX idx_users_organization ON users(organization_id);

-- ============================================
-- TROUBLESHOOTING TREES (Decision Trees)
-- ============================================

CREATE TABLE troubleshooting_trees (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
```

```
    organization_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    slug VARCHAR(100) NOT NULL,
    version INTEGER DEFAULT 1,
    is_published BOOLEAN DEFAULT false,
    published_at TIMESTAMPTZ,
    mermaid_source TEXT,  -- Original Mermaid diagram
    root_step_id UUID,
    equipment_type VARCHAR(100),
    fault_codes VARCHAR(50)[],
    metadata JSONB DEFAULT '{}',
    created_by UUID REFERENCES users(id),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    UNIQUE(organization_id, slug)
);

CREATE INDEX idx_trees_org ON troubleshooting_trees(organization_id);
CREATE INDEX idx_trees_published ON troubleshooting_trees(is_published) WHERE is_published = true;
CREATE INDEX idx_trees_equipment ON troubleshooting_trees(equipment_type);

-- ============================================
-- TROUBLESHOOTING STEPS (Tree Nodes)
-- ============================================

CREATE TYPE step_type AS ENUM ('question', 'action', 'solution', 'escalation', 'redirect');

CREATE TABLE troubleshooting_steps (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    tree_id UUID NOT NULL REFERENCES troubleshooting_trees(id) ON DELETE CASCADE,
    parent_step_id UUID REFERENCES troubleshooting_steps(id) ON DELETE CASCADE,

    -- Step content
    step_type step_type NOT NULL DEFAULT 'question',
    title VARCHAR(500) NOT NULL,
    content TEXT,
    content_html TEXT,

    -- Media attachments
    images JSONB DEFAULT '[]',      -- [{url, alt, caption}]
    videos JSONB DEFAULT '[]',      -- [{url, thumbnail, duration}]
    attachments JSONB DEFAULT '[]', -- [{url, filename, mime_type}]
```

```
    -- Navigation buttons
    buttons JSONB DEFAULT '[]',     -- [{id, label, next_step_id, style}]

    -- Solution fields
    solution_summary TEXT,
    solution_actions JSONB DEFAULT '[]',

    -- Safety & requirements
    safety_warning TEXT,
    tools_required VARCHAR(100)[],
    estimated_time_minutes INTEGER,

    -- Ordering
    position INTEGER DEFAULT 0,
    is_terminal BOOLEAN DEFAULT false,
    metadata JSONB DEFAULT '{}',

    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_steps_tree ON troubleshooting_steps(tree_id);
CREATE INDEX idx_steps_parent ON troubleshooting_steps(parent_step_id);

ALTER TABLE troubleshooting_trees
    ADD CONSTRAINT fk_root_step
    FOREIGN KEY (root_step_id) REFERENCES troubleshooting_steps(id);

-- ============================================
-- USER SESSIONS (Troubleshooting Sessions)
-- ============================================

CREATE TYPE session_status AS ENUM ('active', 'completed', 'abandoned', 'escalated');

CREATE TABLE user_sessions (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    tree_id UUID NOT NULL REFERENCES troubleshooting_trees(id),

    -- Session state
    current_step_id UUID REFERENCES troubleshooting_steps(id),
    status session_status DEFAULT 'active',

    -- Telegram context
```

```
    telegram_chat_id BIGINT,
    telegram_message_id BIGINT,

    -- Equipment context
    equipment_id UUID,
    equipment_serial VARCHAR(255),
    fault_code VARCHAR(50),

    -- Timing
    started_at TIMESTAMPTZ DEFAULT NOW(),
    completed_at TIMESTAMPTZ,
    last_activity_at TIMESTAMPTZ DEFAULT NOW(),

    -- Outcome
    resolution_step_id UUID REFERENCES troubleshooting_steps(id),
    is_proven_path BOOLEAN DEFAULT false,
    was_helpful BOOLEAN,
    feedback_text TEXT,

    -- Path tracking
    path_sequence UUID[] DEFAULT '{}',  -- Ordered step IDs
    path_hash VARCHAR(64),              -- For proven path matching
    total_steps INTEGER DEFAULT 0,
    total_duration_seconds INTEGER,

    context JSONB DEFAULT '{}',
    metadata JSONB DEFAULT '{}',

    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_sessions_user ON user_sessions(user_id);
CREATE INDEX idx_sessions_tree ON user_sessions(tree_id);
CREATE INDEX idx_sessions_status ON user_sessions(status);
CREATE INDEX idx_sessions_active ON user_sessions(user_id, status) WHERE status = 'active';
CREATE INDEX idx_sessions_proven ON user_sessions(is_proven_path) WHERE is_proven_path = true;

-- ============================================
-- JOURNEY EVENTS (Every tap/interaction)
-- ============================================

CREATE TYPE event_type AS ENUM (
    'session_started',
```

```
    'step_viewed',
    'button_clicked',
    'back_navigation',
    'restart',
    'media_viewed',
    'help_clicked',
    'solution_reached',
    'escalation_triggered',
    'feedback_submitted',
    'session_completed',
    'session_abandoned'
);

CREATE TABLE journey_events (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    session_id UUID NOT NULL REFERENCES user_sessions(id) ON DELETE CASCADE,
    user_id UUID NOT NULL REFERENCES users(id),

    -- Event details
    event_type event_type NOT NULL,
    step_id UUID REFERENCES troubleshooting_steps(id),
    step_number INTEGER,

    -- Button click data
    button_id VARCHAR(100),
    button_label VARCHAR(255),

    -- Navigation context
    from_step_id UUID REFERENCES troubleshooting_steps(id),
    to_step_id UUID REFERENCES troubleshooting_steps(id),

    -- Timing
    event_timestamp TIMESTAMPTZ DEFAULT NOW(),
    time_on_step_ms INTEGER,

    -- Telegram context
    telegram_message_id BIGINT,
    telegram_callback_query_id VARCHAR(100),

    metadata JSONB DEFAULT '{}',
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_events_session ON journey_events(session_id);
```

```
CREATE INDEX idx_events_user ON journey_events(user_id);
CREATE INDEX idx_events_type ON journey_events(event_type);
CREATE INDEX idx_events_timestamp ON journey_events(event_timestamp);
CREATE INDEX idx_events_step ON journey_events(step_id);

-- ============================================
-- PROVEN PATHS (Successful resolution patterns)
-- ============================================

CREATE TABLE proven_paths (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    tree_id UUID NOT NULL REFERENCES troubleshooting_trees(id),
    equipment_type VARCHAR(100),
    fault_code VARCHAR(50),

    path_sequence UUID[] NOT NULL,
    path_hash VARCHAR(64) UNIQUE NOT NULL,

    success_count INTEGER DEFAULT 1,
    total_attempts INTEGER DEFAULT 1,
    success_rate DECIMAL(5,2),
    avg_completion_seconds INTEGER,

    confidence_tier VARCHAR(20) DEFAULT 'unverified',  -- gold, silver, bronze, unverified
    is_verified BOOLEAN DEFAULT false,

    first_recorded_at TIMESTAMPTZ DEFAULT NOW(),
    last_recorded_at TIMESTAMPTZ DEFAULT NOW(),

    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_proven_tree ON proven_paths(tree_id);
CREATE INDEX idx_proven_equipment ON proven_paths(equipment_type);
CREATE INDEX idx_proven_fault ON proven_paths(fault_code);
CREATE INDEX idx_proven_confidence ON proven_paths(confidence_tier);

-- ============================================
-- PIPELINE STORIES (Agent Pipeline Tracking)
-- ============================================

CREATE TYPE pipeline_status AS ENUM ('pending', 'running', 'completed', 'failed', 'cancelled');

CREATE TABLE pipeline_stories (
```

```
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),

    -- Context
    tree_id UUID REFERENCES troubleshooting_trees(id),
    user_id UUID REFERENCES users(id),

    -- Pipeline definition
    pipeline_name VARCHAR(255) NOT NULL,
    pipeline_version VARCHAR(50) DEFAULT '1.0',

    -- Execution state
    status pipeline_status DEFAULT 'pending',
    current_agent VARCHAR(255),

    -- Input/Output
    input_data JSONB NOT NULL DEFAULT '{}',
    final_output JSONB,

    -- Timing
    started_at TIMESTAMPTZ,
    completed_at TIMESTAMPTZ,
    duration_ms INTEGER,

    -- Error handling
    error_message TEXT,
    error_details JSONB,
    retry_count INTEGER DEFAULT 0,

    trigger_source VARCHAR(100),
    metadata JSONB DEFAULT '{}',

    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_pipelines_status ON pipeline_stories(status);
CREATE INDEX idx_pipelines_tree ON pipeline_stories(tree_id);

-- ============================================
-- AGENT OUTPUTS (Individual Agent Results)
-- ============================================

CREATE TYPE agent_status AS ENUM ('pending', 'running', 'success', 'failed', 'skipped');
```

```sql
CREATE TABLE agent_outputs (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    pipeline_id UUID NOT NULL REFERENCES pipeline_stories(id) ON DELETE CASCADE,

    -- Agent identification
    agent_name VARCHAR(255) NOT NULL,  -- 'designer', 'architect', 'engineer', 'tester', 'deployer'
    agent_version VARCHAR(50) DEFAULT '1.0',
    execution_order INTEGER NOT NULL,

    -- Execution state
    status agent_status DEFAULT 'pending',

    -- Input/Output
    input_data JSONB DEFAULT '{}',
    output_data JSONB,

    -- LLM metrics
    model_name VARCHAR(100),
    prompt_tokens INTEGER,
    completion_tokens INTEGER,
    total_tokens INTEGER,

    -- Timing
    started_at TIMESTAMPTZ,
    completed_at TIMESTAMPTZ,
    duration_ms INTEGER,

    -- Error handling
    error_message TEXT,
    error_details JSONB,

    -- Approval tracking
    requires_approval BOOLEAN DEFAULT false,
    approval_status VARCHAR(20),  -- 'pending', 'approved', 'rejected'
    approved_by UUID REFERENCES users(id),
    approved_at TIMESTAMPTZ,
    approval_notes TEXT,

    raw_response JSONB,
    metadata JSONB DEFAULT '{}',

    created_at TIMESTAMPTZ DEFAULT NOW()
);
```

```
CREATE INDEX idx_agent_outputs_pipeline ON agent_outputs(pipeline_id);
CREATE INDEX idx_agent_outputs_agent ON agent_outputs(agent_name);
CREATE INDEX idx_agent_outputs_status ON agent_outputs(status);

-- ============================================
-- ROW-LEVEL SECURITY
-- ============================================

ALTER TABLE troubleshooting_trees ENABLE ROW LEVEL SECURITY;
ALTER TABLE troubleshooting_steps ENABLE ROW LEVEL SECURITY;
ALTER TABLE user_sessions ENABLE ROW LEVEL SECURITY;
ALTER TABLE journey_events ENABLE ROW LEVEL SECURITY;

CREATE ROLE app_user;

CREATE POLICY org_isolation_trees ON troubleshooting_trees
    FOR ALL TO app_user
    USING (organization_id = current_setting('app.current_org_id', true)::uuid);

-- ============================================
-- HELPER FUNCTIONS
-- ============================================

CREATE OR REPLACE FUNCTION set_tenant_context(org_id UUID)
RETURNS void AS $$
BEGIN
    PERFORM set_config('app.current_org_id', org_id::text, false);
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION update_updated_at()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Apply triggers
CREATE TRIGGER update_organizations_timestamp BEFORE UPDATE ON organizations
    FOR EACH ROW EXECUTE FUNCTION update_updated_at();
CREATE TRIGGER update_users_timestamp BEFORE UPDATE ON users
    FOR EACH ROW EXECUTE FUNCTION update_updated_at();
CREATE TRIGGER update_trees_timestamp BEFORE UPDATE ON troubleshooting_trees
```

```
    FOR EACH ROW EXECUTE FUNCTION update_updated_at();
CREATE TRIGGER update_sessions_timestamp BEFORE UPDATE ON user_sessions
    FOR EACH ROW EXECUTE FUNCTION update_updated_at();
```

Acceptance criteria for database

- [ ] **AC-DB-1**: Schema deploys successfully to Neon PostgreSQL
- [ ] **AC-DB-2**: Branch creation via API completes in < 2 seconds
- [ ] **AC-DB-3**: Preview branches auto-expire after 7 days
- [ ] **AC-DB-4**: RLS policies enforce organization isolation
- [ ] **AC-DB-5**: Pooled connections (via -pooler suffix) handle 1000+ concurrent users
- [ ] **AC-DB-6**: Tree traversal queries execute in < 50ms

---

6. User journey tracking and analytics

Journey event logging

Every interaction is captured in the `journey_events` table:

```javascript
```

```
async function logJourneyEvent(sessionId, userId, eventData) {
  const { event_type, step_id, button_id, button_label, from_step_id, to_step_id } = eventData;

  // Calculate time on previous step
  const lastEvent = await db.query(`
    SELECT event_timestamp FROM journey_events
    WHERE session_id = $1
    ORDER BY event_timestamp DESC LIMIT 1
  `, [sessionId]);

  const timeOnStepMs = lastEvent.rows.length > 0
    ? Date.now() - new Date(lastEvent.rows[0].event_timestamp).getTime()
    : null;

  await db.query(`
    INSERT INTO journey_events
      (session_id, user_id, event_type, step_id, button_id, button_label,
       from_step_id, to_step_id, time_on_step_ms, step_number)
    VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9,
      (SELECT COALESCE(MAX(step_number), 0) + 1 FROM journey_events WHERE session_id = $1))
  `, [sessionId, userId, event_type, step_id, button_id, button_label,
      from_step_id, to_step_id, timeOnStepMs]);
}
```

Proven path identification

```
sql
```

```sql
-- Identify proven paths from successful sessions
WITH completed_paths AS (
  SELECT
    s.tree_id,
    s.equipment_serial,
    s.fault_code,
    s.path_sequence,
    MD5(ARRAY_TO_STRING(s.path_sequence::text[], '|')) as path_hash,
    EXTRACT(EPOCH FROM (s.completed_at - s.started_at)) as duration_seconds
  FROM user_sessions s
  WHERE s.status = 'completed'
    AND s.resolution_step_id IS NOT NULL
),
path_stats AS (
  SELECT
    tree_id,
    path_sequence,
    path_hash,
    COUNT(*) as success_count,
    AVG(duration_seconds) as avg_duration,
    COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (PARTITION BY tree_id) as success_rate
  FROM completed_paths
  GROUP BY tree_id, path_sequence, path_hash
)
INSERT INTO proven_paths (tree_id, path_sequence, path_hash, success_count,
                          avg_completion_seconds, success_rate, confidence_tier)
SELECT
  tree_id,
  path_sequence,
  path_hash,
  success_count,
  avg_duration::integer,
  success_rate,
  CASE
    WHEN success_count >= 50 AND success_rate >= 85 THEN 'gold'
    WHEN success_count >= 20 AND success_rate >= 75 THEN 'silver'
    WHEN success_count >= 10 AND success_rate >= 70 THEN 'bronze'
    ELSE 'unverified'
  END
FROM path_stats
WHERE success_count >= 5
ON CONFLICT (path_hash) DO UPDATE SET
```

```
    success_count = proven_paths.success_count + EXCLUDED.success_count,
    last_recorded_at = NOW();
```

Flow visualization generation

Generate Mermaid diagrams from journey data:

```javascript
```

```
async function generateJourneyMermaid(treeId, dateRange) {
  // Aggregate path data
  const pathData = await db.query(`
    WITH step_transitions AS (
      SELECT
        from_step_id,
        to_step_id,
        COUNT(*) as transition_count
      FROM journey_events je
      JOIN user_sessions us ON je.session_id = us.id
      WHERE us.tree_id = $1
        AND je.event_type = 'button_clicked'
        AND je.event_timestamp BETWEEN $2 AND $3
      GROUP BY from_step_id, to_step_id
    ),
    step_visits AS (
      SELECT step_id, COUNT(*) as visit_count
      FROM journey_events je
      JOIN user_sessions us ON je.session_id = us.id
      WHERE us.tree_id = $1
        AND je.event_type = 'step_viewed'
      GROUP BY step_id
    )
    SELECT
      ts.id, ts.title, ts.step_type,
      sv.visit_count,
      json_agg(json_build_object(
        'to', st.to_step_id,
        'count', st.transition_count
      )) as transitions
    FROM troubleshooting_steps ts
    LEFT JOIN step_visits sv ON ts.id = sv.step_id
    LEFT JOIN step_transitions st ON ts.id = st.from_step_id
    WHERE ts.tree_id = $1
    GROUP BY ts.id, ts.title, ts.step_type, sv.visit_count
  `, [treeId, dateRange.start, dateRange.end]);

  // Generate Mermaid
  let mermaid = 'flowchart TD\n';

  for (const step of pathData.rows) {
    const shape = step.step_type === 'question' ? ['{', '}'] :
                  step.step_type === 'solution' ? ['([', '])'] : ['[', ']'];
```

```
    mermaid += `    ${step.id}${shape[0]}"${step.title}<br/>(${step.visit_count || 0} visits)"${shape[1]}\n`;
  }

  mermaid += '\n';

  for (const step of pathData.rows) {
    if (step.transitions) {
      for (const t of step.transitions.filter(t => t.to)) {
        const thickness = t.count > 100 ? '==>' : t.count > 20 ? '-->' : '-.->';
        mermaid += `    ${step.id} ${thickness}|${t.count}| ${t.to}\n`;
      }
    }
  }

  // Add styling for proven paths
  mermaid += '\n    classDef proven fill:#90EE90,stroke:#228B22\n';
  mermaid += '    classDef dropoff fill:#FFB6C1,stroke:#DC143C\n';

  return mermaid;
}
```

Admin dashboard metrics

```sql
```

```sql
-- Daily KPI summary for dashboard
SELECT
  DATE(started_at) as date,
  COUNT(*) as total_sessions,
  COUNT(*) FILTER (WHERE status = 'completed') as completed,
  COUNT(*) FILTER (WHERE status = 'abandoned') as abandoned,
  COUNT(*) FILTER (WHERE status = 'escalated') as escalated,
  ROUND(100.0 * COUNT(*) FILTER (WHERE status = 'completed') / NULLIF(COUNT(*), 0), 2) as completion_rate,
  ROUND(AVG(total_duration_seconds) / 60.0, 2) as avg_duration_minutes,
  COUNT(*) FILTER (WHERE is_proven_path = true) as proven_path_sessions
FROM user_sessions
WHERE started_at >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY DATE(started_at)
ORDER BY date DESC;

-- Drop-off analysis by step
SELECT
  ts.id,
  ts.title,
  COUNT(DISTINCT je.session_id) as sessions_reached,
  COUNT(DISTINCT je.session_id) FILTER (
    WHERE EXISTS (
      SELECT 1 FROM journey_events je2
      WHERE je2.session_id = je.session_id
      AND je2.event_timestamp > je.event_timestamp
    )
  ) as sessions_continued,
  ROUND(100.0 * (
    COUNT(DISTINCT je.session_id) -
    COUNT(DISTINCT je.session_id) FILTER (WHERE EXISTS (
      SELECT 1 FROM journey_events je2
      WHERE je2.session_id = je.session_id
      AND je2.event_timestamp > je.event_timestamp
    ))
  ) / NULLIF(COUNT(DISTINCT je.session_id), 0), 2) as drop_off_rate
FROM journey_events je
JOIN troubleshooting_steps ts ON je.step_id = ts.id
WHERE je.event_type = 'step_viewed'
GROUP BY ts.id, ts.title
ORDER BY drop_off_rate DESC;
```

Acceptance criteria for analytics

- [ ] **AC-AN-1**: Every button tap creates journey_event record
- [ ] **AC-AN-2**: time_on_step_ms calculated from previous event timestamp
- [ ] **AC-AN-3**: Proven paths identified with gold/silver/bronze tiers
- [ ] **AC-AN-4**: Mermaid flow diagram generated from aggregated journey data
- [ ] **AC-AN-5**: Admin dashboard displays: completion rate, avg duration, drop-off steps
- [ ] **AC-AN-6**: Session recordings tie to user telegram_id and equipment/fault codes

---

7. Mermaid diagram generation

Rendering API integration

Use **Mermaid.ink** for generating diagram images:

```javascript
```

```
const BASE64 = require('base64-url');

function getMermaidImageUrl(mermaidCode, options = {}) {
  const { format = 'png', theme = 'default', bgColor = 'white' } = options;

  // Base64 encode the mermaid code
  const encoded = BASE64.encode(mermaidCode);

  // Build URL
  const baseUrl = format === 'svg'
    ? `https://mermaid.ink/svg/${encoded}`
    : `https://mermaid.ink/img/${encoded}`;

  const params = new URLSearchParams();
  if (format !== 'svg') params.set('type', format);
  if (theme !== 'default') params.set('theme', theme);
  if (bgColor !== 'white') params.set('bgColor', bgColor.replace('#', ''));

  return params.toString() ? `${baseUrl}?${params}` : baseUrl;
}

// For Telegram delivery
async function sendMermaidDiagram(bot, chatId, mermaidCode, caption) {
  const imageUrl = getMermaidImageUrl(mermaidCode, { format: 'png', bgColor: 'FFFFFF' });

  // Fetch image and send as photo
  const response = await fetch(imageUrl);
  const imageBuffer = await response.buffer();

  await bot.sendPhoto(chatId, imageBuffer, {
    caption: caption,
    parse_mode: 'HTML'
  });
}
```

Database to Mermaid conversion

javascript

```
async function generateMermaidFromTree(treeId) {
  const steps = await db.query(`
    WITH RECURSIVE tree AS (
      SELECT id, title, step_type, buttons, parent_step_id, 0 as depth
      FROM troubleshooting_steps
      WHERE tree_id = $1 AND parent_step_id IS NULL

      UNION ALL

      SELECT s.id, s.title, s.step_type, s.buttons, s.parent_step_id, t.depth + 1
      FROM troubleshooting_steps s
      JOIN tree t ON s.parent_step_id = t.id
    )
    SELECT * FROM tree ORDER BY depth, id
  `, [treeId]);

  let mermaid = 'flowchart TD\n';
  const edges = [];

  // Generate nodes
  for (const step of steps.rows) {
    const shape = step.step_type === 'question' ? ['{', '}'] :
                  step.step_type === 'solution' ? ['([', '])'] :
                  step.step_type === 'escalation' ? ['[[', ']]'] : ['[', ']'];

    const safeTitle = step.title.replace(/"/g, "'");
    mermaid += `    ${step.id}${shape[0]}"${safeTitle}"${shape[1]}\n`;

    // Collect edges from buttons
    if (step.buttons) {
      for (const btn of step.buttons) {
        edges.push({
          from: step.id,
          to: btn.next_step_id,
          label: btn.label
        });
      }
    }
  }

  mermaid += '\n';

  // Generate edges
```

```
  for (const edge of edges) {
    if (edge.label) {
      mermaid += `    ${edge.from} -->|${edge.label}| ${edge.to}\n`;
    } else {
      mermaid += `    ${edge.from} --> ${edge.to}\n`;
    }
  }

  return mermaid;
}
```

Acceptance criteria for diagrams

- [ ] **AC-DIA-1**: Mermaid.ink URLs render correctly for diagrams up to 10KB
- [ ] **AC-DIA-2**: Diagram images display in Telegram with proper resolution
- [ ] **AC-DIA-3**: Database records convert to valid Mermaid syntax
- [ ] **AC-DIA-4**: Journey visualization shows visit counts and transition frequencies
- [ ] **AC-DIA-5**: Proven paths highlighted with green styling

---

8. Integration points

API endpoints summary

Endpoint	Method	Purpose
`/auth/telegram/callback`	GET/POST	Handle Telegram Login Widget callback
`/api/pipeline/start`	POST	Trigger new content generation pipeline
`/api/pipeline/:id/approve`	POST	Approve pipeline stage
`/api/trees`	GET	List troubleshooting trees
`/api/trees/:id`	GET	Get tree with steps
`/api/sessions/:userId/active`	GET	Get user's active session
`/api/analytics/dashboard`	GET	Dashboard metrics
`/api/analytics/journey/:treeId`	GET	Journey visualization data
`/webhook/telegram`	POST	Telegram Bot webhook

Telegram Bot webhook handler

javascript

```javascript
// Webhook endpoint for Telegram Bot
app.post('/webhook/telegram', async (req, res) => {
  const update = req.body;

  try {
    if (update.callback_query) {
      await handleCallbackQuery(update.callback_query);
    } else if (update.message) {
      await handleMessage(update.message);
    }

    res.sendStatus(200);
  } catch (error) {
    console.error('Webhook error:', error);
    res.sendStatus(200); // Always return 200 to Telegram
  }
});
```

n8n webhook integration

javascript

```javascript
// n8n webhook payload structure
{
  "pipelineRequest": {
    "type": "new_tree",
    "equipmentType": "HVAC-Chiller",
    "faultCodes": ["E001", "E002"],
    "description": "Chiller not cooling",
    "requestedBy": {
      "userId": "uuid",
      "telegramId": 123456789
    }
  },
  "approvalCallbackUrl": "https://rivetpro.app/api/pipeline/{id}/approve",
  "notificationChatId": -100123456789
}
```

9. Phased implementation plan

Phase 1: Foundation (Weeks 1-2)

Dependencies: None

1. **Database setup**
    - Deploy Neon PostgreSQL project
    - Execute schema DDL
    - Configure branch management
    - Set up connection pooling
2. **Authentication**
    - Create Telegram Bot via BotFather
    - Configure domain with /setdomain
    - Implement login widget on web
    - Build backend hash verification
    - Create JWT session management
3. **Basic Bot**
    - Set up webhook endpoint
    - Implement message handling
    - Create inline keyboard renderer

Phase 2: Troubleshooting Core (Weeks 3-4)

Dependencies: Phase 1 complete

1. **Tree management**
    - Build Mermaid parser
    - Implement tree CRUD API
    - Create step navigation logic
2. **Interactive delivery**
    - Implement step display with media
    - Build callback query handler
    - Add back/restart navigation
    - Implement session state tracking

3. **Journey tracking**
    - Log all events to database
    - Calculate time-on-step metrics
    - Build path sequence tracking

Phase 3: Pipeline Agents (Weeks 5-6)

Dependencies: Phase 2 complete

1. **n8n setup**
    - Deploy n8n instance
    - Configure PostgreSQL connection
    - Set up Telegram integration
2. **Agent workflows**
    - Create Designer Agent workflow
    - Create Architect Agent workflow
    - Create Engineer Agent workflow
    - Create Tester Agent workflow
    - Create Deployer Agent workflow
3. **Orchestration**
    - Build parent orchestrator workflow
    - Implement approval gates
    - Configure error handling
    - Generate backlog.md

Phase 4: Analytics & Admin (Weeks 7-8)

Dependencies: Phase 3 complete

1. **Analytics engine**
    - Implement proven path identification
    - Build journey aggregation queries
    - Create materialized views
2. **Visualization**
    - Generate Mermaid from journeys
    - Integrate Mermaid.ink rendering

- Build flow diagram component

3. **Admin dashboard**
    - Display KPI metrics
    - Show drop-off analysis
    - List proven paths
    - Monitor pipeline status

Phase 5: Polish & Scale (Weeks 9-10)

Dependencies: Phase 4 complete

1. **Claude fallback**
    - Implement unknown issue handler
    - Build "save guide" workflow
    - Train on successful patterns
2. **Performance optimization**
    - Add database indexes
    - Implement caching layer
    - Optimize query patterns
3. **Multi-tenancy**
    - Enable RLS policies
    - Test organization isolation
    - Configure per-org branches

---

10. Technical specifications summary

Component	Technology	Version/Notes
Database	Neon PostgreSQL	Serverless, branching enabled
Backend	Node.js	v20 LTS
Workflows	n8n	Self-hosted or Cloud
AI	Claude API	claude-3-5-sonnet-20241022

Component	Technology	Version/Notes
Bot Framework	node-telegram-bot-api	v0.66+
Diagrams	Mermaid.ink	API v10+
Auth	Telegram Login Widget	Widget v22
Session	JWT	7-day access, 30-day refresh

Environment variables

```bash
# Telegram
BOT_TOKEN=your_bot_token
APPROVAL_CHAT_ID=-100123456789

# Neon Database
DATABASE_URL=postgres://user:pass@ep-xxx-pooler.neon.tech/rivetpro
DATABASE_URL_DIRECT=postgres://user:pass@ep-xxx.neon.tech/rivetpro
NEON_API_KEY=your_neon_api_key
NEON_PROJECT_ID=your_project_id

# Claude
ANTHROPIC_API_KEY=your_anthropic_key

# n8n
N8N_WEBHOOK_URL=https://n8n.rivetpro.app/webhook
N8N_API_KEY=your_n8n_api_key

# Application
JWT_SECRET=your_jwt_secret
REFRESH_SECRET=your_refresh_secret
APP_URL=https://rivetpro.app
```

Conclusion

This PRD provides a complete implementation blueprint for RIVET Pro's multi-agent pipeline system. The architecture combines **Telegram's ubiquitous mobile presence** with **n8n's workflow orchestration**, **Neon's database branching** for safe development, and **Claude's AI capabilities** for content generation and fallback

troubleshooting. The journey tracking system enables continuous improvement by identifying proven paths and visualizing user behavior.

Key innovations include using Mermaid diagrams as the universal authoring format (convertible to both database records and visual documentation), Telegram inline buttons for tap-through troubleshooting delivery, and a five-agent pipeline with human approval gates for quality control.

The phased implementation plan ensures dependencies are satisfied in order, with the foundation (database, auth, basic bot) enabling subsequent phases. Each phase has clear acceptance criteria that the Ralph autonomous agent system can verify during implementation.