AfriMail Pro - Complete Implementation Guide & Actor Analysis

1. SYSTEM ACTORS & ROLES ANALYSIS

1.1 Primary Actors

Super Administrator (System Owner)

- **Profile**: Momo Godi Yvan (Project Owner)
- **Primary Role**: System oversight and strategic management
- **Functionalities**:
  - Full system access and configuration
  - User account management and billing oversight
  - Platform analytics and performance monitoring
  - Email domain configuration management
  - Template library management
  - System security and compliance monitoring
  - Integration management (SMTP providers, payment gateways)
  - Support escalation handling

Company Administrator

- **Profile**: PME owners, CEOs, Business owners
- **Primary Role**: Account management and team coordination
- **Functionalities**:
  - Company account setup and configuration
  - Team member invitation and role assignment
  - Billing and subscription management
  - Email domain configuration for company
  - Company-wide analytics and reporting
  - Integration settings (CRM, e-commerce platforms)
  - Compliance and data management
  - Brand customization (logo, colors, templates)

Marketing Manager

- **Profile**: Marketing directors, campaign managers
- **Primary Role**: Campaign strategy and execution
- **Functionalities**:
  - Campaign planning and strategy development
  - Advanced segmentation and targeting
  - A/B testing setup and analysis
  - ROI tracking and performance optimization
  - Team collaboration and approval workflows
  - Budget allocation and spend tracking
  - Competitor analysis and benchmarking
  - Marketing automation setup

Email Marketing Specialist

- **Profile**: Marketing executives, content creators
- **Primary Role**: Daily campaign operations
- **Functionalities**:
  - Email template creation and customization
  - Content writing and personalization
  - Contact list management and segmentation
  - Campaign scheduling and execution
  - Performance monitoring and reporting
  - A/B testing execution
  - Customer journey mapping
  - Deliverability optimization

Sales Representative

- **Profile**: Sales team members, account managers
- **Primary Role**: Lead nurturing and conversion
- **Functionalities**:
  - Lead scoring and qualification
  - Follow-up email sequences
  - Customer onboarding campaigns
  - Sales pipeline integration
  - Contact interaction tracking
  - Deal-specific email campaigns
  - Customer feedback collection
  - Referral program management

Freelance Consultant

- **Profile**: Independent marketing consultants, agencies
- **Primary Role**: Multi-client management
- **Functionalities**:
  - Multi-client account management
  - White-label solution access
  - Client reporting and analytics
  - Template library for multiple brands
  - Billing management per client
  - Performance benchmarking across clients
  - API access for custom integrations
  - Client onboarding and training

Developer/Integrator

- **Profile**: Technical team members, IT specialists
- **Primary Role**: Technical integration and customization
- **Functionalities**:
  - API integration and management
  - Custom field creation and management
  - Webhook configuration
  - Third-party service connections
  - Data migration and import/export
  - Custom analytics setup
  - Security configuration
  - Technical troubleshooting

1.2 Secondary Actors

End Customer/Recipient

- **Role**: Email recipients and interaction generators
- **Actions**:
  - Email opening and reading
  - Link clicking and engagement
  - Unsubscribe requests
  - Preference management
  - Feedback and surveys
  - Social sharing
  - Purchase actions

SMTP Service Provider

- **Role**: Email delivery infrastructure
- **Actions**:
  - Email routing and delivery
  - Bounce and complaint handling
  - Delivery status reporting
  - IP reputation management
  - Security scanning

2. EMAIL DOMAIN CONFIGURATION SOLUTION

2.1 Current Challenge Analysis

Your concern about email domain configuration is valid and critical for a multi-tenant SaaS platform. Here's the comprehensive solution:

2.2 Proposed Multi-Domain Email Solution

Option 1: User-Configurable SMTP Settings (Recommended)

python

```python
# Domain Configuration Model
class EmailDomainConfig(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    domain_name = models.CharField(max_length=100)  # e.g., "mycompany.com"
    from_email = models.EmailField()  # e.g., "marketing@mycompany.com"
    from_name = models.CharField(max_length=100)  # e.g., "MyCompany Marketing"

    # SMTP Configuration
    smtp_provider = models.CharField(max_length=50, choices=SMTP_PROVIDERS)
    smtp_host = models.CharField(max_length=100)
    smtp_port = models.IntegerField(default=587)
    smtp_username = models.CharField(max_length=100)
    smtp_password = models.CharField(max_length=100)  # Encrypted
    use_tls = models.BooleanField(default=True)

    # Verification Status
    domain_verified = models.BooleanField(default=False)
    spf_verified = models.BooleanField(default=False)
    dkim_verified = models.BooleanField(default=False)
    dmarc_verified = models.BooleanField(default=False)

    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)

# Email Sending Service
class EmailSender:
    def __init__(self, user):
        self.user = user
        self.domain_config = self.get_user_domain_config()

    def get_user_domain_config(self):
        # Get user's primary domain config or fallback to default
        return EmailDomainConfig.objects.filter(
            user=self.user,
            is_active=True
        ).first() or self.get_default_config()

    def send_email(self, recipients, subject, content):
        if self.domain_config:
            # Use user's configured SMTP
            return self.send_via_custom_smtp(recipients, subject, content)
        else:
            # Fallback to platform default
            return self.send_via_default_smtp(recipients, subject, content)
```

Option 2: Hybrid Approach (Default + Custom)

Features:

- **Default Domain**: All users start with `noreply@afrimailpro.com`
- **Custom Domain Upgrade**: Premium feature for custom domains
- **Easy Configuration**: User-friendly domain setup wizard
- **Automatic Verification**: DNS verification process

2.3 Implementation Strategy

Phase 1: Default System Domain

```python
# settings.py
DEFAULT_EMAIL_CONFIG = {
    'host': 'smtp.afrimailpro.com',
    'port': 587,
    'username': 'system@afrimailpro.com',
    'password': 'encrypted_password',
    'from_email': 'noreply@afrimailpro.com',
    'from_name': 'AfriMail Pro'
}
```

Phase 2: User Domain Configuration

python

```python
# Domain Setup Wizard
class DomainSetupWizard:
    def step1_basic_info(self):
        """Collect domain and email preferences"""
        pass

    def step2_smtp_config(self):
        """SMTP provider selection and configuration"""
        pass

    def step3_dns_verification(self):
        """Guide user through DNS setup"""
        pass

    def step4_testing(self):
        """Send test emails and verify delivery"""
        pass
```

Phase 3: Advanced Features

- Multiple domain support per user
- Domain delegation for team members
- Automatic DNS monitoring
- Deliverability optimization

3. STEP-BY-STEP IMPLEMENTATION GUIDE

3.1 Phase 1: Foundation & Landing Page (Weeks 1-2)

Step 1: Project Setup

```bash
# Project Structure
afrimail-pro/
├── backend/
│   ├── afrimail/
│   │   ├── settings/
│   │   ├── apps/
│   │   │   ├── authentication/
│   │   │   ├── campaigns/
│   │   │   ├── contacts/
│   │   │   ├── templates/
│   │   │   ├── analytics/
│   │   │   └── email_service/
│   │   └── utils/
│   ├── requirements.txt
│   ├── Dockerfile
│   └── docker-compose.yml
├── frontend/
│   ├── static/
│   ├── templates/
│   └── assets/
└── docs/
```

Step 2: Landing Page Development

```html
<!-- Landing Page Structure -->
<main class="landing-page">
    <!-- Hero Section -->
    <section class="hero-section">
        <h1>Connectez l'Afrique, Un Email à la Fois</h1>
        <p>AfriMail Pro: Votre Partenaire Marketing Digital</p>
        <div class="cta-buttons">
            <button class="btn-primary">Essai Gratuit 14 Jours</button>
            <button class="btn-secondary">Voir la Démo</button>
        </div>
    </section>

    <!-- Features Section -->
    <section class="features-section">
        <div class="feature-cards">
            <div class="feature-card">
                <h3>Tarifs Adaptés à l'Afrique</h3>
                <p>70% moins cher que les solutions internationales</p>
            </div>
            <!-- More feature cards -->
        </div>
    </section>

    <!-- Social Proof -->
    <section class="testimonials">
        <!-- Customer testimonials -->
    </section>

    <!-- Pricing -->
    <section class="pricing-section">
        <!-- Pricing tiers -->
    </section>
</main>
```

Step 3: User Authentication System

python

```python
# Custom User Model
class CustomUser(AbstractUser):
    email = models.EmailField(unique=True)
    company = models.CharField(max_length=100)
    phone = models.CharField(max_length=20)
    country = models.CharField(max_length=50)
    industry = models.CharField(max_length=50)
    company_size = models.CharField(max_length=20)

    # Onboarding
    onboarding_completed = models.BooleanField(default=False)
    trial_started = models.DateTimeField(null=True)
    trial_ends = models.DateTimeField(null=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name', 'company']

# Registration Process
class UserRegistrationView(CreateView):
    def post(self, request):
        # 1. Validate form data
        # 2. Create user account
        # 3. Send welcome email
        # 4. Start trial period
        # 5. Redirect to onboarding
        pass
```

3.2 Phase 2: Core Functionality (Weeks 3-8)

Step 4: Contact Management System

python

```python
# Contact Model with Advanced Features
class Contact(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    email = models.EmailField()
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    company = models.CharField(max_length=100, blank=True)
    phone = models.CharField(max_length=20, blank=True)

    # Geographic Data
    country = models.CharField(max_length=50, blank=True)
    city = models.CharField(max_length=50, blank=True)
    timezone = models.CharField(max_length=50, blank=True)

    # Engagement Metrics
    engagement_score = models.FloatField(default=0)
    last_engagement = models.DateTimeField(null=True)
    total_opens = models.IntegerField(default=0)
    total_clicks = models.IntegerField(default=0)

    # Behavioral Data
    preferred_send_time = models.TimeField(null=True)
    preferred_frequency = models.CharField(max_length=20, default='weekly')
    interests = models.JSONField(default=list)
    purchase_history = models.JSONField(default=list)

    # Status
    is_subscribed = models.BooleanField(default=True)
    subscription_source = models.CharField(max_length=50)
    tags = models.ManyToManyField('ContactTag', blank=True)

    class Meta:
        unique_together = ['user', 'email']
        indexes = [
            models.Index(fields=['user', 'engagement_score']),
            models.Index(fields=['email']),
            models.Index(fields=['last_engagement']),
        ]

# Contact Import Service
class ContactImporter:
    def __init__(self, user, file_path):
        self.user = user
        self.file_path = file_path
    def import_contacts(self):
        # 1. Detect file format
        # 2. Parse and validate data
        # 3. Check for duplicates
        # 4. Enrich contact data
        # 5. Batch create contacts
        # 6. Generate import report
        pass
```

Step 5: Email Template System

python

```python
# Template Management
class EmailTemplate(models.Model):
    name = models.CharField(max_length=100)
    category = models.CharField(max_length=50)
    industry = models.CharField(max_length=50)
    html_content = models.TextField()
    css_styles = models.TextField()
    variables = models.JSONField(default=list)

    # Metadata
    thumbnail = models.ImageField(upload_to='template_thumbnails/')
    is_premium = models.BooleanField(default=False)
    usage_count = models.IntegerField(default=0)
    rating = models.FloatField(default=0)

    # Responsive Design
    mobile_optimized = models.BooleanField(default=True)
    dark_mode_support = models.BooleanField(default=False)

# Drag & Drop Editor
class TemplateEditor:
    def __init__(self, template_id=None):
        self.template = self.get_template(template_id)

    def add_block(self, block_type, position, content):
        """Add content block to template"""
        pass

    def update_block(self, block_id, content):
        """Update existing block"""
        pass

    def delete_block(self, block_id):
        """Remove block from template"""
        pass

    def generate_preview(self, device='desktop'):
        """Generate template preview"""
        pass
```

Step 6: Campaign Management

python

```python
# Campaign Model
class Campaign(models.Model):
    CAMPAIGN_TYPES = [
        ('newsletter', 'Newsletter'),
        ('promotional', 'Promotional'),
        ('transactional', 'Transactional'),
        ('automated', 'Automated'),
    ]

    CAMPAIGN_STATUS = [
        ('draft', 'Draft'),
        ('scheduled', 'Scheduled'),
        ('sending', 'Sending'),
        ('sent', 'Sent'),
        ('paused', 'Paused'),
        ('completed', 'Completed'),
    ]

    user = models.ForeignKey(User, on_delete=models.CASCADE)
    name = models.CharField(max_length=200)
    subject = models.CharField(max_length=200)
    preview_text = models.CharField(max_length=150, blank=True)

    # Content
    html_content = models.TextField()
    text_content = models.TextField()
    template = models.ForeignKey(EmailTemplate, null=True, blank=True)

    # Targeting
    segments = models.ManyToManyField('ContactSegment', blank=True)
    exclude_segments = models.ManyToManyField('ContactSegment', blank=True, related_name='exclu

    # Scheduling
    campaign_type = models.CharField(max_length=20, choices=CAMPAIGN_TYPES)
    status = models.CharField(max_length=20, choices=CAMPAIGN_STATUS, default='draft')
    scheduled_at = models.DateTimeField(null=True, blank=True)
    send_immediately = models.BooleanField(default=False)

    # A/B Testing
    is_ab_test = models.BooleanField(default=False)
    ab_test_percentage = models.IntegerField(default=50)
    ab_winner_criteria = models.CharField(max_length=20, default='open_rate')

    # Analytics
    recipients_count = models.IntegerField(default=0)
    recipients_count = models.IntegerField(default=0)
    sent_count = models.IntegerField(default=0)
    delivered_count = models.IntegerField(default=0)
    opened_count = models.IntegerField(default=0)
    clicked_count = models.IntegerField(default=0)
    unsubscribed_count = models.IntegerField(default=0)
    bounced_count = models.IntegerField(default=0)

# Campaign Sender Service
class CampaignSender:
    def __init__(self, campaign):
        self.campaign = campaign

    def prepare_send(self):
        """Prepare campaign for sending"""
        # 1. Validate campaign content
        # 2. Get recipient list
        # 3. Personalize content
        # 4. Queue email jobs
        pass

    def send_batch(self, contacts_batch):
        """Send emails to a batch of contacts"""
        pass

    def handle_bounces(self):
        """Process bounce notifications"""
        pass
```

3.3 Phase 3: Advanced Features (Weeks 9-12)

Step 7: Marketing Automation

python

```python
# Automation Triggers
class AutomationTrigger(models.Model):
    TRIGGER_TYPES = [
        ('welcome', 'New Subscriber'),
        ('birthday', 'Birthday'),
        ('anniversary', 'Anniversary'),
        ('abandoned_cart', 'Abandoned Cart'),
        ('post_purchase', 'Post Purchase'),
        ('inactive', 'Inactive Subscriber'),
        ('behavioral', 'Behavioral Trigger'),
    ]

    name = models.CharField(max_length=100)
    trigger_type = models.CharField(max_length=20, choices=TRIGGER_TYPES)
    conditions = models.JSONField()
    delay_amount = models.IntegerField(default=0)
    delay_unit = models.CharField(max_length=10, default='hours')

    is_active = models.BooleanField(default=True)

# Automation Flow
class AutomationFlow(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    description = models.TextField()

    trigger = models.ForeignKey(AutomationTrigger, on_delete=models.CASCADE)

    # Flow Control
    is_active = models.BooleanField(default=True)
    start_date = models.DateTimeField()
    end_date = models.DateTimeField(null=True, blank=True)

    # Performance
    subscribers_count = models.IntegerField(default=0)
    completed_count = models.IntegerField(default=0)
    conversion_rate = models.FloatField(default=0)

# Automation Processor
class AutomationProcessor:
    def process_triggers(self):
        """Check and process all active triggers"""
        pass

    def execute_flow(self, contact, automation_flow):
```

```
    def execute_flow(self, contact, automation_flow):
        """Execute automation flow for specific contact"""
        pass

    def track_automation_performance(self):
        """Update automation analytics"""
        pass
```

Step 8: Analytics Dashboard

python

```python
# Analytics Models
class CampaignAnalytics(models.Model):
    campaign = models.OneToOneField(Campaign, on_delete=models.CASCADE)

    # Delivery Metrics
    delivery_rate = models.FloatField(default=0)
    bounce_rate = models.FloatField(default=0)

    # Engagement Metrics
    open_rate = models.FloatField(default=0)
    click_rate = models.FloatField(default=0)
    click_to_open_rate = models.FloatField(default=0)
    unsubscribe_rate = models.FloatField(default=0)

    # Advanced Metrics
    forward_rate = models.FloatField(default=0)
    social_share_rate = models.FloatField(default=0)
    conversion_rate = models.FloatField(default=0)
    revenue_generated = models.DecimalField(max_digits=10, decimal_places=2, default=0)
    roi = models.FloatField(default=0)

    # Time-based Analysis
    best_send_time = models.TimeField(null=True)
    peak_engagement_day = models.CharField(max_length=10)

    updated_at = models.DateTimeField(auto_now=True)

# Analytics Dashboard
class AnalyticsDashboard:
    def __init__(self, user, date_range=None):
        self.user = user
        self.date_range = date_range or self.get_default_range()

    def get_overview_metrics(self):
        """Get high-level performance metrics"""
        return {
            'total_campaigns': self.get_campaign_count(),
            'total_subscribers': self.get_subscriber_count(),
            'avg_open_rate': self.get_average_open_rate(),
            'avg_click_rate': self.get_average_click_rate(),
            'total_revenue': self.get_total_revenue(),
            'roi': self.get_overall_roi(),
        }

    def get_performance_trends(self):
```

```
    def get_performance_trends(self):
        """Get performance trends over time"""
        pass

    def get_audience_insights(self):
        """Get audience behavior insights"""
        pass
```

3.4 Phase 4: Integration & Optimization (Weeks 13-16)

Step 9: API Development

python

```python
# REST API for Integrations
from rest_framework import viewsets, permissions
from rest_framework.decorators import action
from rest_framework.response import Response

class ContactViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        return Contact.objects.filter(user=self.request.user)

    @action(detail=False, methods=['post'])
    def bulk_import(self, request):
        """Bulk import contacts via API"""
        pass

    @action(detail=True, methods=['post'])
    def add_tags(self, request, pk=None):
        """Add tags to contact"""
        pass

class CampaignViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticated]

    @action(detail=True, methods=['post'])
    def send(self, request, pk=None):
        """Send campaign via API"""
        pass

    @action(detail=True, methods=['get'])
    def analytics(self, request, pk=None):
        """Get campaign analytics"""
        pass

# Webhook System
class WebhookManager:
    def register_webhook(self, user, event_type, url):
        """Register webhook for user events"""
        pass

    def trigger_webhook(self, event_type, data):
        """Trigger registered webhooks"""
        pass
```

Step 10: Mobile Optimization

```css
/* Mobile-First Responsive Design */
.dashboard {
    display: grid;
    grid-template-columns: 1fr;
    gap: 1rem;
    padding: 1rem;
}

@media (min-width: 768px) {
    .dashboard {
        grid-template-columns: 250px 1fr;
        padding: 2rem;
    }
}

@media (min-width: 1024px) {
    .dashboard {
        grid-template-columns: 300px 1fr 300px;
    }
}

/* Touch-Friendly Controls */
.btn {
    min-height: 44px;
    min-width: 44px;
    padding: 0.75rem 1.5rem;
}

/* Progressive Web App Features */
.offline-indicator {
    position: fixed;
    top: 0;
    left: 50%;
    transform: translateX(-50%);
    background: #f59e0b;
    color: white;
    padding: 0.5rem 1rem;
    border-radius: 0 0 0.5rem 0.5rem;
    display: none;
}

.offline .offline-indicator {
    display: block;
}
```

4. EMAIL DOMAIN CONFIGURATION INTERFACE

4.1 User Interface for Domain Setup

html

```html
<!-- Domain Configuration Wizard -->
<div class="domain-setup-wizard">
    <!-- Step 1: Domain Selection -->
    <div class="wizard-step" data-step="1">
        <h3>Configure Your Email Domain</h3>
        <div class="domain-options">
            <label class="option-card">
                <input type="radio" name="domain_type" value="default">
                <div class="option-content">
                    <h4>Use AfriMail Pro Domain (Free)</h4>
                    <p>Send emails from: noreply@afrimailpro.com</p>
                    <span class="badge">Recommended for getting started</span>
                </div>
            </label>

            <label class="option-card">
                <input type="radio" name="domain_type" value="custom">
                <div class="option-content">
                    <h4>Use Your Company Domain (Premium)</h4>
                    <p>Send emails from: marketing@yourcompany.com</p>
                    <span class="badge premium">Better deliverability & branding</span>
                </div>
            </label>
        </div>
    </div>

    <!-- Step 2: Custom Domain Configuration -->
    <div class="wizard-step" data-step="2" style="display: none;">
        <h3>Configure Your SMTP Settings</h3>
        <form class="domain-form">
            <div class="form-group">
                <label>Your Domain</label>
                <input type="text" name="domain" placeholder="yourcompany.com">
            </div>

            <div class="form-group">
                <label>From Email</label>
                <input type="email" name="from_email" placeholder="marketing@yourcompany.com">
            </div>

            <div class="form-group">
                <label>From Name</label>
                <input type="text" name="from_name" placeholder="Your Company Name">
            </div>
            <div class="smtp-provider-selection">
                <label>Choose SMTP Provider</label>
                <select name="smtp_provider">
                    <option value="gmail">Gmail (G Suite)</option>
                    <option value="outlook">Outlook 365</option>
                    <option value="sendgrid">SendGrid</option>
                    <option value="mailgun">Mailgun</option>
                    <option value="custom">Custom SMTP</option>
                </select>
            </div>

            <div class="smtp-settings" id="smtp-custom" style="display: none;">
                <div class="form-row">
                    <div class="form-group">
                        <label>SMTP Host</label>
                        <input type="text" name="smtp_host" placeholder="smtp.yourprovider.com"
                    </div>
                    <div class="form-group">
                        <label>Port</label>
                        <input type="number" name="smtp_port" value="587">
                    </div>
                </div>

                <div class="form-row">
                    <div class="form-group">
                        <label>Username</label>
                        <input type="text" name="smtp_username">
                    </div>
                    <div class="form-group">
                        <label>Password</label>
                        <input type="password" name="smtp_password">
                    </div>
                </div>

                <div class="form-group">
                    <label class="checkbox">
                        <input type="checkbox" name="use_tls" checked>
                        Use TLS Encryption
                    </label>
                </div>
            </div>
        </form>
    </div>

    <!-- Step 3: DNS Verification -->
    <div class="wizard-step" data-step="3" style="display: none;">
```

```
        <h3>Verify Your Domain</h3>
        <div class="dns-instructions">
            <p>Add these DNS records to verify your domain:</p>

            <div class="dns-record">
                <strong>SPF Record:</strong>
                <code>v=spf1 include:afrimailpro.com ~all</code>
                <button class="copy-btn">Copy</button>
            </div>

            <div class="dns-record">
                <strong>DKIM Record:</strong>
                <code>selector._domainkey.yourcompany.com</code>
                <button class="copy-btn">Copy</button>
            </div>

            <div class="dns-record">
                <strong>DMARC Record:</strong>
                <code>v=DMARC1; p=quarantine; rua=mailto:dmarc@afrimailpro.com</code>
                <button class="copy-btn">Copy</button>
            </div>
        </div>

        <div class="verification-status">
            <div class="verification-item">
                <span class="status pending">⏳</span>
                <span>SPF Record</span>
            </div>
            <div class="verification-item">
                <span class="status pending">⏳</span>
                <span>DKIM Record</span>
            </div>
            <div class="verification-item">
                <span class="status pending">⏳</span>
                <span>DMARC Record</span>
            </div>
        </div>

        <button class="btn-primary" onclick="checkDNSRecords()">
            Check DNS Records
        </button>
    </div>

    <!-- Step 4: Test & Complete -->
    <div class="wizard-step" data-step="4" style="display: none;">
        <h3>Test Your Configuration</h3>
        <div class="test-email-form">
            <div class="form-group">
                <label>Send test email to:</label>
                <input type="email" name="test_email" placeholder="your-email@example.com">
            </div>
            <button class="btn-primary" onclick="sendTestEmail()">
                Send Test Email
            </button>
        </div>

        <div class="test-results" id="test-results" style="display: none;">
            <!-- Test results will be displayed here -->
        </div>
    </div>
</div>
```

4.2 Backend Domain Management

python

```python
# Domain Configuration Service
class DomainConfigurationService:
    def __init__(self, user):
        self.user = user

    def create_domain_config(self, domain_data):
        """Create new domain configuration"""
        config = EmailDomainConfig.objects.create(
            user=self.user,
            domain_name=domain_data['domain_name'],
            from_email=domain_data['from_email'],
            from_name=domain_data['from_name'],
            smtp_provider=domain_data['smtp_provider'],
            smtp_host=domain_data['smtp_host'],
            smtp_port=domain_data['smtp_port'],
            smtp_username=domain_data['smtp_username'],
            smtp_password=self.encrypt_password(domain_data['smtp_password']),
            use_tls=domain_data.get('use_tls', True)
        )

        # Start DNS verification process
        self.initiate_dns_verification(config)
        return config

    def verify_dns_records(self, config):
        """Verify DNS records for domain"""
        verification_results = {
            'spf_verified': self.check_spf_record(config.domain_name),
            'dkim_verified': self.check_dkim_record(config.domain_name),
            'dmarc_verified': self.check_dmarc_record(config.domain_name)
        }

        # Update configuration
        config.spf_verified = verification_results['spf_verified']
        config.dkim_verified = verification_results['dkim_verified']
        config.dmarc_verified = verification_results['dmarc_verified']
        config.domain_verified = all(verification_results.values())
        config.save()

        return verification_results

    def test_smtp_connection(self, config):
        """Test SMTP connection and send test email"""
        try:
            smtp_client = self.get_smtp_client(config)
            smtp_client = self.get_smtp_client(config)
            smtp_client.connect()

            # Send test email
            test_result = self.send_test_email(smtp_client, config)
            smtp_client.quit()

            return {
                'success': True,
                'message': 'SMTP connection successful',
                'test_email_sent': test_result
            }
        except Exception as e:
            return {
                'success': False,
                'message': f'SMTP connection failed: {str(e)}'
            }

    def get_user_sending_domains(self):
        """Get all configured domains for user"""
        return EmailDomainConfig.objects.filter(
            user=self.user,
            is_active=True
        ).order_by('-domain_verified', 'created_at')

# Enhanced Email Sending Service
class EnhancedEmailSender:
    def __init__(self, user, campaign=None):
        self.user = user
        self.campaign = campaign
        self.domain_config = self.select_best_domain_config()

    def select_best_domain_config(self):
        """Select the best domain configuration for sending"""
        # Priority: Verified custom domain > Unverified custom domain > Default
        configs = EmailDomainConfig.objects.filter(
            user=self.user,
            is_active=True
        ).order_by('-domain_verified', '-created_at')

        if configs.exists():
            return configs.first()
        else:
            # Return default platform configuration
            return self.get_default_platform_config()

    def get_default_platform_config(self):
        """Get default platform email configuration"""
        return {
            'domain_name': 'afrimailpro.com',
            'from_email': 'noreply@afrimailpro.com',
            'from_name': 'AfriMail Pro',
            'smtp_host': settings.DEFAULT_SMTP_HOST,
            'smtp_port': settings.DEFAULT_SMTP_PORT,
            'smtp_username': settings.DEFAULT_SMTP_USERNAME,
            'smtp_password': settings.DEFAULT_SMTP_PASSWORD,
            'use_tls': True
        }

    def send_campaign_emails(self, recipients, subject, content):
        """Send campaign emails using appropriate domain"""
        if self.domain_config and hasattr(self.domain_config, 'domain_verified'):
            if self.domain_config.domain_verified:
                return self.send_via_custom_domain(recipients, subject, content)
            else:
                # Notify user about unverified domain and use default
                self.notify_unverified_domain()
                return self.send_via_default_domain(recipients, subject, content)
        else:
            return self.send_via_default_domain(recipients, subject, content)

    def send_via_custom_domain(self, recipients, subject, content):
        """Send emails using user's custom domain"""
        try:
            # Use user's SMTP configuration
            smtp_config = {
                'host': self.domain_config.smtp_host,
                'port': self.domain_config.smtp_port,
                'username': self.domain_config.smtp_username,
                'password': self.decrypt_password(self.domain_config.smtp_password),
                'use_tls': self.domain_config.use_tls
            }

            from_email = f"{self.domain_config.from_name} <{self.domain_config.from_email}>"

            return self.send_bulk_emails(recipients, subject, content, from_email, smtp_config)

        except Exception as e:
            # Fallback to default domain on error
            self.log_smtp_error(e)
            return self.send_via_default_domain(recipients, subject, content)

    def send_via_default_domain(self, recipients, subject, content):
```

~~~
        """Send emails using platform default domain"""
        default_config = self.get_default_platform_config()
        from_email = f"{default_config['from_name']} <{default_config['from_email']}>"

        smtp_config = {
            'host': default_config['smtp_host'],
            'port': default_config['smtp_port'],
            'username': default_config['smtp_username'],
            'password': default_config['smtp_password'],
            'use_tls': default_config['use_tls']
        }

        return self.send_bulk_emails(recipients, subject, content, from_email, smtp_config)

## 5. DETAILED IMPLEMENTATION ROADMAP

### 5.1 Sprint-by-Sprint Breakdown

#### **Sprint 1-2: Foundation & Landing (Weeks 1-4)**

**Sprint 1 Goals:**
- Complete project setup and infrastructure
- Develop responsive landing page
- Implement basic user authentication
- Set up deployment pipeline

**Sprint 1 Tasks:**
```python
# Task Breakdown
SPRINT_1_TASKS = {
    'infrastructure': [
        'Setup Django project with Docker',
        'Configure PostgreSQL database',
        'Setup Redis for caching and queues',
        'Configure CI/CD with GitHub Actions',
        'Setup staging environment'
    ],
    'frontend': [
        'Design system with Tailwind CSS',
        'Responsive landing page',
        'User registration/login forms',
        'Email verification system',
        'Password reset functionality'
    ],
    'backend': [
        'Custom user model',
        'Authentication views and APIs',
~~~

```
        'Email service integration',
        'Basic admin interface',
        'Security middleware setup'
    ]
}
```

Sprint 2 Goals:

- Complete onboarding flow
- Implement trial system
- Basic dashboard structure
- Payment integration setup

Sprint 2 Tasks:

```python
SPRINT_2_TASKS = {
    'onboarding': [
        'Multi-step onboarding wizard',
        'Company profile setup',
        'Trial activation system',
        'Welcome email sequence',
        'Onboarding progress tracking'
    ],
    'dashboard': [
        'Dashboard layout and navigation',
        'User profile management',
        'Basic settings interface',
        'Help and support system',
        'Notification system'
    ],
    'payments': [
        'Subscription model setup',
        'Mobile money integration (MTN, Orange)',
        'Billing dashboard',
        'Invoice generation',
        'Payment reminder system'
    ]
}
```

Sprint 3-4: Core Contact Management (Weeks 5-8)

Sprint 3 Goals:

- Complete contact management system
- File import functionality
- Basic segmentation
- Contact validation and deduplication

Sprint 3 Implementation:

python

```python
# Contact Import Service Implementation
class ContactImportService:
    SUPPORTED_FORMATS = ['csv', 'xlsx', 'xls', 'vcf', 'json']

    def __init__(self, user, file_path, file_format):
        self.user = user
        self.file_path = file_path
        self.file_format = file_format
        self.import_stats = {
            'total_rows': 0,
            'valid_contacts': 0,
            'invalid_contacts': 0,
            'duplicates': 0,
            'imported': 0,
            'errors': []
        }

    def process_import(self):
        """Main import processing method"""
        try:
            # Step 1: Parse file
            raw_data = self.parse_file()

            # Step 2: Validate and clean data
            cleaned_data = self.validate_and_clean(raw_data)

            # Step 3: Check for duplicates
            unique_data = self.handle_duplicates(cleaned_data)

            # Step 4: Enrich contact data
            enriched_data = self.enrich_contacts(unique_data)

            # Step 5: Bulk create contacts
            imported_contacts = self.bulk_create_contacts(enriched_data)

            # Step 6: Generate import report
            return self.generate_import_report(imported_contacts)

        except Exception as e:
            self.import_stats['errors'].append(str(e))
            return self.import_stats

    def parse_file(self):
        """Parse uploaded file based on format"""
        if self.file_format == 'csv':
            return self.parse_csv()
        elif self.file_format in ['xlsx', 'xls']:
            return self.parse_excel()
        elif self.file_format == 'vcf':
            return self.parse_vcard()
        elif self.file_format == 'json':
            return self.parse_json()
        else:
            raise ValueError(f"Unsupported file format: {self.file_format}")

    def validate_and_clean(self, raw_data):
        """Validate email addresses and clean data"""
        cleaned_contacts = []

        for row in raw_data:
            contact = self.validate_contact_row(row)
            if contact:
                cleaned_contacts.append(contact)
            else:
                self.import_stats['invalid_contacts'] += 1

        return cleaned_contacts

    def validate_contact_row(self, row):
        """Validate individual contact row"""
        # Email validation
        email = row.get('email', '').strip().lower()
        if not self.is_valid_email(email):
            return None

        # Phone number validation and formatting
        phone = row.get('phone', '').strip()
        if phone:
            phone = self.format_african_phone_number(phone)

        # Name cleaning
        first_name = row.get('first_name', '').strip().title()
        last_name = row.get('last_name', '').strip().title()

        return {
            'email': email,
            'first_name': first_name,
            'last_name': last_name,
            'company': row.get('company', '').strip(),
            'phone': phone,
            'country': row.get('country', '').strip(),
            'city': row.get('city', '').strip(),
            'tags': self.parse_tags(row.get('tags', '')),
            'custom_fields': self.extract_custom_fields(row)
        }

    def format_african_phone_number(self, phone):
        """Format phone numbers for African countries"""
        # Remove all non-digit characters
        digits_only = re.sub(r'\D', '', phone)

        # African country codes mapping
        country_codes = {
            '237': 'CM',  # Cameroon
            '234': 'NG',  # Nigeria
            '254': 'KE',  # Kenya
            '27': 'ZA',   # South Africa
            '233': 'GH',  # Ghana
            '225': 'CI',  # Ivory Coast
        }

        # Format based on length and country code
        if len(digits_only) >= 10:
            for code, country in country_codes.items():
                if digits_only.startswith(code):
                    return f"+{digits_only}"

            # If no country code detected, assume local number
            return f"+237{digits_only}" if len(digits_only) == 9 else f"+{digits_only}"

        return phone  # Return original if can't format

# Contact Segmentation System
class AdvancedSegmentation:
    def __init__(self, user):
        self.user = user

    def create_segment(self, name, conditions):
        """Create new contact segment"""
        segment = ContactSegment.objects.create(
            user=self.user,
            name=name,
            conditions=conditions
        )

        # Calculate initial contact count
        segment.contact_count = self.calculate_segment_size(conditions)
        segment.save()

        return segment

    def calculate_segment_size(self, conditions):
        """Calculate how many contacts match segment conditions"""
        queryset = Contact.objects.filter(user=self.user)

        for condition in conditions:
            field = condition['field']
            operator = condition['operator']
            value = condition['value']

            if operator == 'equals':
                queryset = queryset.filter(**{field: value})
            elif operator == 'contains':
                queryset = queryset.filter(**{f"{field}__icontains": value})
            elif operator == 'starts_with':
                queryset = queryset.filter(**{f"{field}__istartswith": value})
            elif operator == 'greater_than':
                queryset = queryset.filter(**{f"{field}__gt": value})
            elif operator == 'less_than':
                queryset = queryset.filter(**{f"{field}__lt": value})
            elif operator == 'in_list':
                queryset = queryset.filter(**{f"{field}__in": value})
            elif operator == 'not_equals':
                queryset = queryset.exclude(**{field: value})

        return queryset.count()

    def get_segment_contacts(self, segment):
        """Get all contacts that match segment conditions"""
        return self.calculate_segment_size(segment.conditions, return_queryset=True)

# Contact Engagement Scoring
class EngagementScorer:
    def __init__(self):
        self.scoring_weights = {
            'email_opens': 2,
            'email_clicks': 5,
            'website_visits': 3,
            'form_submissions': 8,
            'purchases': 15,
            'social_shares': 4,
            'referrals': 10,
            'recency_factor': 1.5  # More recent activities score higher
        }
    def calculate_engagement_score(self, contact):
        """Calculate engagement score for contact"""
        score = 0

        # Get contact interactions from last 90 days
        recent_interactions = ContactInteraction.objects.filter(
            contact=contact,
            timestamp__gte=timezone.now() - timedelta(days=90)
        )

        for interaction in recent_interactions:
            base_score = self.scoring_weights.get(interaction.type, 1)

            # Apply recency factor
            days_ago = (timezone.now() - interaction.timestamp).days
            recency_multiplier = max(0.1, 1 - (days_ago / 90))

            score += base_score * recency_multiplier

        # Normalize score to 0-100 range
        normalized_score = min(100, score)

        # Update contact engagement score
        contact.engagement_score = normalized_score
        contact.last_engagement = recent_interactions.first().timestamp if recent_interactions.
        contact.save()

        return normalized_score
```

Sprint 4 Goals:

- Email template system
- Drag & drop editor
- Template marketplace
- Preview functionality

Sprint 5-6: Campaign Management (Weeks 9-12)

Sprint 5 Implementation:

python

```python
# Advanced Campaign Builder
class CampaignBuilder:
    def __init__(self, user):
        self.user = user
        self.campaign = None

    def create_campaign(self, campaign_data):
        """Create new email campaign"""
        self.campaign = Campaign.objects.create(
            user=self.user,
            name=campaign_data['name'],
            subject=campaign_data['subject'],
            preview_text=campaign_data.get('preview_text', ''),
            campaign_type=campaign_data['type'],
            html_content=campaign_data.get('html_content', ''),
            text_content=campaign_data.get('text_content', '')
        )

        # Set up A/B testing if requested
        if campaign_data.get('enable_ab_test'):
            self.setup_ab_test(campaign_data['ab_test_config'])

        return self.campaign

    def setup_ab_test(self, ab_config):
        """Setup A/B testing for campaign"""
        self.campaign.is_ab_test = True
        self.campaign.ab_test_percentage = ab_config.get('percentage', 50)
        self.campaign.ab_winner_criteria = ab_config.get('criteria', 'open_rate')

        # Create variant campaigns
        variant_a = self.create_campaign_variant('A', ab_config['variant_a'])
        variant_b = self.create_campaign_variant('B', ab_config['variant_b'])

        self.campaign.save()
        return variant_a, variant_b

    def personalize_content(self, content, contact):
        """Personalize email content for specific contact"""
        personalizations = {
            '{{first_name}}': contact.first_name or 'Valued Customer',
            '{{last_name}}': contact.last_name or '',
            '{{company}}': contact.company or '',
            '{{email}}': contact.email,
            '{{phone}}': contact.phone or ''
            '{{phone}}': contact.phone or '',
            '{{country}}': contact.country or '',
            '{{city}}': contact.city or ''
        }

        # Add custom field personalizations
        for field_name, field_value in contact.custom_fields.items():
            personalizations[f'{{{{{field_name}}}}}'] = str(field_value)

        # Apply personalizations
        personalized_content = content
        for placeholder, value in personalizations.items():
            personalized_content = personalized_content.replace(placeholder, value)

        return personalized_content

    def schedule_campaign(self, send_time, time_zone='Africa/Douala'):
        """Schedule campaign for future sending"""
        # Convert to UTC for storage
        local_tz = pytz.timezone(time_zone)
        utc_send_time = local_tz.localize(send_time).astimezone(pytz.UTC)

        self.campaign.scheduled_at = utc_send_time
        self.campaign.status = 'scheduled'
        self.campaign.save()

        # Queue the campaign for sending
        from .tasks import send_scheduled_campaign
        send_scheduled_campaign.apply_async(
            args=[self.campaign.id],
            eta=utc_send_time
        )

# Smart Send Time Optimization
class SendTimeOptimizer:
    def __init__(self, user):
        self.user = user

    def get_optimal_send_time(self, segment=None):
        """Calculate optimal send time based on recipient behavior"""
        # Analyze historical open rates by time and day
        if segment:
            contacts = segment.get_contacts()
        else:
            contacts = Contact.objects.filter(user=self.user)

        # Get engagement data by hour and day
        engagement_data = self.analyze_engagement_patterns(contacts)

        # Find peak engagement times
        optimal_times = self.find_peak_engagement_times(engagement_data)

        return optimal_times

    def analyze_engagement_patterns(self, contacts):
        """Analyze when contacts are most likely to engage"""
        engagement_by_hour = defaultdict(list)
        engagement_by_day = defaultdict(list)

        for contact in contacts:
            interactions = ContactInteraction.objects.filter(
                contact=contact,
                type__in=['email_open', 'email_click'],
                timestamp__gte=timezone.now() - timedelta(days=90)
            )

            for interaction in interactions:
                local_time = interaction.timestamp.astimezone(
                    pytz.timezone(contact.timezone or 'Africa/Douala')
                )

                hour = local_time.hour
                day = local_time.strftime('%A')

                engagement_by_hour[hour].append(1)
                engagement_by_day[day].append(1)

        return {
            'by_hour': {hour: sum(engagements) for hour, engagements in engagement_by_hour.iten
            'by_day': {day: sum(engagements) for day, engagements in engagement_by_day.items()
        }

# Campaign Performance Tracker
class CampaignTracker:
    def __init__(self, campaign):
        self.campaign = campaign

    def track_email_sent(self, contact, email_log_id):
        """Track when email is sent"""
        EmailLog.objects.create(
            campaign=self.campaign,
            contact=contact,
            status='sent',
            sent_at=timezone.now(),
            email_log_id=email_log_id
        )

        # Update campaign stats
        self.campaign.sent_count += 1
        self.campaign.save()

    def track_email_opened(self, contact, user_agent=None, ip_address=None):
        """Track email open event"""
        email_log = EmailLog.objects.filter(
            campaign=self.campaign,
            contact=contact,
            status='sent'
        ).first()

        if email_log and not email_log.opened_at:
            email_log.opened_at = timezone.now()
            email_log.save()

            # Update campaign stats
            self.campaign.opened_count += 1
            self.campaign.save()

            # Update contact engagement
            ContactInteraction.objects.create(
                contact=contact,
                type='email_open',
                campaign=self.campaign,
                timestamp=timezone.now(),
                metadata={
                    'user_agent': user_agent,
                    'ip_address': ip_address
                }
            )

            # Update engagement score
            EngagementScorer().calculate_engagement_score(contact)

    def track_link_clicked(self, contact, link_url, user_agent=None):
        """Track link click event"""
        email_log = EmailLog.objects.filter(
            campaign=self.campaign,
            contact=contact
        ).first()

        if email_log:
```

```python
        if not email_log.clicked_at:
            email_log.clicked_at = timezone.now()
            email_log.save()

            # Update campaign stats
            self.campaign.clicked_count += 1
            self.campaign.save()

        # Record click interaction
        ContactInteraction.objects.create(
            contact=contact,
            type='email_click',
            campaign=self.campaign,
            timestamp=timezone.now(),
            metadata={
                'link_url': link_url,
                'user_agent': user_agent
            }
        )

        # Update engagement score
        EngagementScorer().calculate_engagement_score(contact)
```

Sprint 7-8: Analytics & Automation (Weeks 13-16)

Advanced Analytics System:

python

```python
# Comprehensive Analytics Engine
class AnalyticsEngine:
    def __init__(self, user, date_range=None):
        self.user = user
        self.date_range = date_range or self.get_default_date_range()

    def get_campaign_performance_summary(self):
        """Get overall campaign performance metrics"""
        campaigns = Campaign.objects.filter(
            user=self.user,
            sent_at__range=self.date_range,
            status='completed'
        )

        total_sent = campaigns.aggregate(Sum('sent_count'))['sent_count__sum'] or 0
        total_opened = campaigns.aggregate(Sum('opened_count'))['opened_count__sum'] or 0
        total_clicked = campaigns.aggregate(Sum('clicked_count'))['clicked_count__sum'] or 0
        total_unsubscribed = campaigns.aggregate(Sum('unsubscribed_count'))['unsubscribed_count

        return {
            'total_campaigns': campaigns.count(),
            'total_emails_sent': total_sent,
            'average_open_rate': (total_opened / total_sent * 100) if total_sent > 0 else 0,
            'average_click_rate': (total_clicked / total_sent * 100) if total_sent > 0 else 0,
            'average_unsubscribe_rate': (total_unsubscribed / total_sent * 100) if total_sent
            'engagement_trend': self.calculate_engagement_trend(),
            'top_performing_campaigns': self.get_top_campaigns(5),
            'audience_growth': self.calculate_audience_growth()
        }

    def generate_roi_analysis(self):
        """Calculate ROI for campaigns with revenue tracking"""
        roi_data = []

        campaigns_with_revenue = Campaign.objects.filter(
            user=self.user,
            campaignanalytics__revenue_generated__gt=0,
            sent_at__range=self.date_range
        ).select_related('campaignanalytics')

        for campaign in campaigns_with_revenue:
            analytics = campaign.campaignanalytics
            campaign_cost = self.calculate_campaign_cost(campaign)

            roi = ((analytics.revenue_generated - campaign_cost) / campaign_cost * 100) if camp
            roi = ((analytics.revenue_generated - campaign_cost) / campaign_cost * 100) if camp

            roi_data.append({
                'campaign_name': campaign.name,
                'revenue': float(analytics.revenue_generated),
                'cost': campaign_cost,
                'roi': roi,
                'sent_date': campaign.sent_at,
                'recipients': campaign.sent_count
            })

        return sorted(roi_data, key=lambda x: x['roi'], reverse=True)

    def get_audience_insights(self):
        """Analyze audience behavior and preferences"""
        contacts = Contact.objects.filter(user=self.user)

        # Geographic distribution
        geographic_data = contacts.values('country').annotate(
            count=Count('id')
        ).order_by('-count')

        # Engagement levels
        engagement_distribution = {
            'highly_engaged': contacts.filter(engagement_score__gte=70).count(),
            'moderately_engaged': contacts.filter(engagement_score__range=[30, 69]).count(),
            'low_engaged': contacts.filter(engagement_score__lt=30).count(),
            'inactive': contacts.filter(last_engagement__lt=timezone.now() - timedelta(days=90)
        }

        # Device and client analysis
        device_data = self.analyze_device_usage()

        # Best performing content types
        content_performance = self.analyze_content_performance()

        return {
            'geographic_distribution': list(geographic_data),
            'engagement_distribution': engagement_distribution,
            'device_usage': device_data,
            'content_preferences': content_performance,
            'optimal_send_times': self.get_optimal_send_times(),
            'subject_line_analysis': self.analyze_subject_lines()
        }

    def generate_predictive_insights(self):
        """Generate AI-powered predictive insights"""
        insights = []

        # Predict churn risk
        churn_predictions = self.predict_churn_risk()
        if churn_predictions['high_risk_count'] > 0:
            insights.append({
                'type': 'churn_warning',
                'message': f"{churn_predictions['high_risk_count']} contacts are at high risk c
                'recommendation': "Consider sending a re-engagement campaign",
                'priority': 'high'
            })

        # Predict optimal send frequency
        frequency_analysis = self.analyze_send_frequency()
        insights.append({
            'type': 'frequency_optimization',
            'message': f"Your optimal send frequency is {frequency_analysis['optimal_frequency'
            'current_frequency': frequency_analysis['current_frequency'],
            'recommendation': frequency_analysis['recommendation'],
            'priority': 'medium'
        })

        # Identify high-value segments
        valuable_segments = self.identify_valuable_segments()
        for segment in valuable_segments:
            insights.append({
                'type': 'segment_opportunity',
                'message': f"Segment '{segment['name']}' shows high conversion potential",
                'conversion_rate': segment['conversion_rate'],
                'recommendation': f"Increase targeting for {segment['name']} segment",
                'priority': 'medium'
            })

        return insights

# Marketing Automation Engine
class AutomationEngine:
    def __init__(self):
        self.trigger_processors = {
            'welcome': WelcomeSequenceProcessor(),
            'abandoned_cart': AbandonedCartProcessor(),
            'birthday': BirthdayProcessor(),
            'inactive': InactiveSubscriberProcessor(),
            'behavioral': BehavioralTriggerProcessor()
        }
```

```python
    def process_automation_triggers(self):
        """Process all active automation triggers"""
        active_automations = AutomationFlow.objects.filter(
            is_active=True,
            start_date__lte=timezone.now()
        ).select_related('trigger')

        for automation in active_automations:
            processor = self.trigger_processors.get(automation.trigger.trigger_type)
            if processor:
                processor.process_automation(automation)

    def execute_automation_step(self, automation_execution):
        """Execute individual automation step"""
        step = automation_execution.current_step
        contact = automation_execution.contact
        automation = automation_execution.automation

        if step['type'] == 'send_email':
            self.send_automation_email(contact, step, automation)
        elif step['type'] == 'wait':
            self.schedule_next_step(automation_execution, step['delay'])
        elif step['type'] == 'condition':
            self.evaluate_condition(automation_execution, step)
        elif step['type'] == 'add_tag':
            self.add_contact_tag(contact, step['tag'])
        elif step['type'] == 'update_field':
            self.update_contact_field(contact, step['field'], step['value'])

# Behavioral Trigger System
class BehavioralTriggerProcessor:
    def __init__(self):
        self.behavior_patterns = {
            'high_engagement': self.detect_high_engagement,
            'declining_engagement': self.detect_declining_engagement,
```