Complete Development Plan: Planable Clone with Pro Features

1. PROJECT OVERVIEW & ARCHITECTURE

1.1 Core Platform Components

- **Frontend**: React.js with TypeScript for type safety
- **Backend**: Node.js with Express.js and TypeScript
- **Database**: PostgreSQL for relational data, Redis for caching and sessions
- **File Storage**: AWS S3 or similar for media assets
- **Real-time**: Socket.io for live collaboration features
- **Queue System**: Bull Queue with Redis for scheduled posts
- **Authentication**: JWT tokens with refresh token rotation
- **API Integration**: Social media platform APIs (Facebook, Instagram, Twitter, LinkedIn, TikTok, YouTube, Pinterest, Google My Business, Telegram)

1.2 Microservices Architecture

```
├── API Gateway (Express.js)
├── Authentication Service
├── Content Management Service
├── Social Media Integration Service
├── Analytics Service
├── Notification Service
├── File Processing Service
├── Scheduling Service
└── Collaboration Service
```

2. CORE FEATURES BREAKDOWN

2.1 Content Creation & Management

Implementation Steps:

1. **Rich Text Editor Integration**
   - Integrate TipTap or Quill.js for WYSIWYG editing
   - Support for mentions, hashtags, emojis
   - Character count tracking per platform
   - Auto-save functionality every 30 seconds

2. **Multi-Platform Content Composer**
    - Platform-specific templates and constraints
    - Image/video upload with compression
    - Platform-specific preview modes
    - Bulk content creation tools
    - Content versioning system
3. **Media Management**
    - Drag-and-drop file uploads
    - Image editing tools (crop, resize, filters)
    - Video trimming capabilities
    - Media library with tagging and search
    - Auto-optimization for different platforms

2.2 Calendar & Scheduling System

Implementation Steps:

1. **Visual Calendar Interface**
    - Monthly, weekly, daily views
    - Drag-and-drop post scheduling
    - Color coding by platform/campaign
    - Bulk scheduling operations
    - Time zone management
2. **Smart Scheduling Engine**
    - Optimal posting time suggestions
    - Queue management system
    - Auto-rescheduling for failed posts
    - Recurring post templates
    - Bulk import from CSV/Excel
3. **Publishing Automation**
    - Multi-platform simultaneous posting
    - Platform-specific optimization
    - Error handling and retry logic
    - Post status tracking
    - Emergency stop functionality

2.3 Collaboration & Approval Workflow

Implementation Steps:

1. **Multi-Level Approval System**
   - Custom approval workflows
   - Role-based permissions (Creator, Reviewer, Approver, Admin)
   - Sequential and parallel approval flows
   - Approval deadline management
   - Auto-escalation rules
2. **Real-time Collaboration**
   - Live editing with conflict resolution
   - Comment system with threading
   - @mentions and notifications
   - Activity feed and audit logs
   - Version history with rollback
3. **Client Collaboration Portal**
   - Client-only workspace views
   - Simplified approval interface
   - Email notifications for approvals
   - Mobile-friendly approval process
   - Brand safety guidelines integration

2.4 Analytics & Reporting

Implementation Steps:

1. **Cross-Platform Analytics**
   - Unified dashboard for all platforms
   - Engagement metrics aggregation
   - ROI tracking and attribution
   - Competitor analysis tools
   - Custom report builder
2. **Performance Insights**
   - Post performance predictions
   - Optimal posting time analysis
   - Audience insights and demographics

- Content performance comparisons
- Growth tracking and trends

3. **White-label Reporting**
   - Custom branded reports
   - Automated report generation
   - PDF export functionality
   - Client-facing dashboards
   - KPI tracking and alerts

3. TECHNICAL IMPLEMENTATION ROADMAP

Phase 1: Foundation (Weeks 1-4)

Week 1-2: Infrastructure Setup

1. Set up development environment

```bash
# Backend setup
mkdir planable-clone && cd planable-clone
mkdir backend frontend
cd backend
npm init -y
npm install express typescript @types/node @types/express
npm install -D nodemon ts-node
```

2. Database schema design

```sql
```

```sql
-- Core tables structure
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  role VARCHAR(50) NOT NULL,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE workspaces (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL,
  owner_id UUID REFERENCES users(id),
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE posts (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  workspace_id UUID REFERENCES workspaces(id),
  content TEXT NOT NULL,
  platforms JSON NOT NULL,
  scheduled_at TIMESTAMP,
  status VARCHAR(50) DEFAULT 'draft',
  created_by UUID REFERENCES users(id),
  created_at TIMESTAMP DEFAULT NOW()
);
```

3. Authentication system implementation
4. Basic API structure with Express.js

Week 3-4: Core Backend Services

1. User management and workspace creation
2. Basic CRUD operations for posts
3. File upload handling with multer
4. JWT authentication middleware
5. Database connection and ORM setup (Prisma or TypeORM)

Phase 2: Content Management (Weeks 5-8)

Week 5-6: Content Creation System

1. Rich text editor integration
2. Multi-platform content composer

3. Media upload and processing
4. Content validation by platform

Week 7-8: Content Organization

1. Content calendar implementation
2. Drag-and-drop scheduling
3. Content categorization and tagging
4. Search and filtering system

Phase 3: Social Media Integration (Weeks 9-12)

Week 9-10: Platform APIs Integration

1. Facebook/Instagram Graph API integration
2. Twitter API v2 implementation
3. LinkedIn API integration
4. TikTok Business API (if available)

Week 11-12: Publishing System

1. Scheduled posting engine with Bull Queue
2. Error handling and retry logic
3. Post status tracking
4. Platform-specific optimizations

Phase 4: Collaboration Features (Weeks 13-16)

Week 13-14: Approval Workflow

1. Custom approval workflow builder
2. Multi-level approval system
3. Role-based permissions
4. Email notifications

Week 15-16: Real-time Collaboration

1. Socket.io integration for live editing
2. Comment system implementation
3. Activity feed and notifications
4. Version control system

Phase 5: Analytics & Reporting (Weeks 17-20)

Week 17-18: Data Collection

1. Social media metrics aggregation
2. Analytics data processing
3. Database optimization for analytics

Week 19-20: Dashboard & Reports

1. Analytics dashboard creation
2. Custom report builder
3. PDF report generation
4. Data visualization components

Phase 6: Advanced Features (Weeks 21-24)

Week 21-22: AI Features

1. Content optimization suggestions
2. Optimal posting time analysis
3. Hashtag recommendations
4. Content performance predictions

Week 23-24: Enterprise Features

1. White-label branding
2. Custom integrations
3. Advanced security features
4. Audit logs and compliance

4. DETAILED FEATURE SPECIFICATIONS

4.1 User Management & Authentication

typescript

```typescript
// User roles and permissions
enum UserRole {
  SUPER_ADMIN = 'super_admin',
  WORKSPACE_ADMIN = 'workspace_admin',
  CONTENT_MANAGER = 'content_manager',
  CONTENT_CREATOR = 'content_creator',
  REVIEWER = 'reviewer',
  CLIENT = 'client'
}

interface User {
  id: string;
  email: string;
  firstName: string;
  lastName: string;
  role: UserRole;
  permissions: Permission[];
  workspaces: WorkspaceMember[];
  createdAt: Date;
  lastLogin: Date;
}
```

4.2 Content Management System

typescript

```typescript
interface Post {
  id: string;
  workspaceId: string;
  title: string;
  content: PlatformContent[];
  media: MediaAsset[];
  scheduledAt?: Date;
  publishedAt?: Date;
  status: PostStatus;
  platforms: SocialPlatform[];
  tags: string[];
  campaign?: string;
  approvalWorkflow: ApprovalStep[];
  createdBy: string;
  updatedBy: string;
  createdAt: Date;
  updatedAt: Date;
}

interface PlatformContent {
  platform: SocialPlatform;
  text: string;
  hashtags: string[];
  mentions: string[];
  customFields: Record<string, any>;
}
```

4.3 Approval Workflow System

typescript

```
interface ApprovalWorkflow {
  id: string;
  name: string;
  steps: ApprovalStep[];
  workspaceId: string;
  isDefault: boolean;
  conditions: WorkflowCondition[];
}

interface ApprovalStep {
  id: string;
  order: number;
  name: string;
  approvers: User[];
  requiredApprovals: number;
  autoApprove: boolean;
  deadline?: number; // hours
  escalation?: EscalationRule;
}
```

5. DATABASE DESIGN

5.1 Core Tables Schema

sql

```sql
-- Users and Authentication
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  first_name VARCHAR(100),
  last_name VARCHAR(100),
  avatar_url VARCHAR(500),
  role VARCHAR(50) NOT NULL,
  is_active BOOLEAN DEFAULT true,
  email_verified BOOLEAN DEFAULT false,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Workspaces (Team/Client separation)
CREATE TABLE workspaces (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL,
  slug VARCHAR(100) UNIQUE NOT NULL,
  description TEXT,
  logo_url VARCHAR(500),
  owner_id UUID REFERENCES users(id),
  settings JSONB DEFAULT '{}',
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Workspace Members
CREATE TABLE workspace_members (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  workspace_id UUID REFERENCES workspaces(id) ON DELETE CASCADE,
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  role VARCHAR(50) NOT NULL,
  permissions JSONB DEFAULT '{}',
  joined_at TIMESTAMP DEFAULT NOW(),
  UNIQUE(workspace_id, user_id)
);

-- Social Media Accounts
CREATE TABLE social_accounts (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  workspace_id UUID REFERENCES workspaces(id) ON DELETE CASCADE,
  platform VARCHAR(50) NOT NULL,
  account_id VARCHAR(255) NOT NULL,
```

```sql
  account_name VARCHAR(255),
  access_token TEXT,
  refresh_token TEXT,
  token_expires_at TIMESTAMP,
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Posts
CREATE TABLE posts (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  workspace_id UUID REFERENCES workspaces(id) ON DELETE CASCADE,
  title VARCHAR(500),
  content JSONB NOT NULL, -- Platform-specific content
  media_assets JSONB DEFAULT '[]',
  platforms VARCHAR(50)[] NOT NULL,
  scheduled_at TIMESTAMP,
  published_at TIMESTAMP,
  status VARCHAR(50) DEFAULT 'draft',
  tags VARCHAR(100)[],
  campaign VARCHAR(255),
  created_by UUID REFERENCES users(id),
  updated_by UUID REFERENCES users(id),
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Approval Workflows
CREATE TABLE approval_workflows (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  workspace_id UUID REFERENCES workspaces(id) ON DELETE CASCADE,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  steps JSONB NOT NULL,
  conditions JSONB DEFAULT '{}',
  is_default BOOLEAN DEFAULT false,
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Post Approvals
CREATE TABLE post_approvals (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  post_id UUID REFERENCES posts(id) ON DELETE CASCADE,
  workflow_id UUID REFERENCES approval_workflows(id),
```

```
  current_step INTEGER DEFAULT 0,
  status VARCHAR(50) DEFAULT 'pending',
  approved_by JSONB DEFAULT '[]',
  rejected_by JSONB DEFAULT '[]',
  comments JSONB DEFAULT '[]',
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Analytics Data
CREATE TABLE post_analytics (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  post_id UUID REFERENCES posts(id) ON DELETE CASCADE,
  platform VARCHAR(50) NOT NULL,
  platform_post_id VARCHAR(255),
  metrics JSONB NOT NULL,
  collected_at TIMESTAMP DEFAULT NOW()
);

-- Comments and Collaboration
CREATE TABLE post_comments (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  post_id UUID REFERENCES posts(id) ON DELETE CASCADE,
  user_id UUID REFERENCES users(id),
  content TEXT NOT NULL,
  parent_id UUID REFERENCES post_comments(id),
  mentions UUID[],
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);
```

5.2 Indexes for Performance

```sql
-- Performance indexes
CREATE INDEX idx_posts_workspace_id ON posts(workspace_id);
CREATE INDEX idx_posts_status ON posts(status);
CREATE INDEX idx_posts_scheduled_at ON posts(scheduled_at);
CREATE INDEX idx_posts_created_at ON posts(created_at);
CREATE INDEX idx_workspace_members_workspace_id ON workspace_members(workspace_id);
CREATE INDEX idx_workspace_members_user_id ON workspace_members(user_id);
CREATE INDEX idx_post_analytics_post_id ON post_analytics(post_id);
CREATE INDEX idx_post_analytics_collected_at ON post_analytics(collected_at);
```

6. API DESIGN & ENDPOINTS

6.1 RESTful API Structure

typescript

```
// Authentication endpoints
POST /api/auth/register
POST /api/auth/login
POST /api/auth/logout
POST /api/auth/refresh
POST /api/auth/forgot-password
POST /api/auth/reset-password

// User management
GET /api/users/profile
PUT /api/users/profile
GET /api/users/workspaces

// Workspace management
GET /api/workspaces
POST /api/workspaces
GET /api/workspaces/:id
PUT /api/workspaces/:id
DELETE /api/workspaces/:id
GET /api/workspaces/:id/members
POST /api/workspaces/:id/members
PUT /api/workspaces/:id/members/:userId
DELETE /api/workspaces/:id/members/:userId

// Social accounts
GET /api/workspaces/:id/social-accounts
POST /api/workspaces/:id/social-accounts
DELETE /api/workspaces/:id/social-accounts/:accountId
GET /api/social-accounts/:id/auth-url
POST /api/social-accounts/:id/callback

// Posts management
GET /api/workspaces/:id/posts
POST /api/workspaces/:id/posts
GET /api/posts/:id
PUT /api/posts/:id
DELETE /api/posts/:id
POST /api/posts/:id/duplicate
POST /api/posts/:id/schedule
POST /api/posts/:id/publish
POST /api/posts/:id/cancel

// Approval workflow
GET /api/workspaces/:id/approval-workflows
POST /api/workspaces/:id/approval-workflows
PUT /api/approval-workflows/:id
```

```
DELETE /api/approval-workflows/:id
POST /api/posts/:id/submit-for-approval
POST /api/posts/:id/approve
POST /api/posts/:id/reject

// Analytics
GET /api/workspaces/:id/analytics
GET /api/posts/:id/analytics
GET /api/workspaces/:id/reports
POST /api/workspaces/:id/reports/generate

// Media management
POST /api/media/upload
GET /api/media/:id
DELETE /api/media/:id
POST /api/media/:id/edit
```

6.2 WebSocket Events for Real-time Features

typescript

```typescript
// Socket.io events
interface SocketEvents {
  // Collaboration
  'post:edit': (postId: string, changes: Partial<Post>) => void;
  'post:comment': (postId: string, comment: Comment) => void;
  'post:approve': (postId: string, approval: Approval) => void;

  // Notifications
  'notification:new': (notification: Notification) => void;
  'notification:read': (notificationId: string) => void;

  // Live updates
  'post:status-change': (postId: string, status: PostStatus) => void;
  'workspace:member-join': (workspaceId: string, member: User) => void;
}
```

7. FRONTEND IMPLEMENTATION

7.1 React Component Architecture

```
src/
├── components/
│   ├── common/
│   │   ├── Button/
│   │   ├── Modal/
│   │   ├── Form/
│   │   └── Layout/
│   ├── content/
│   │   ├── PostComposer/
│   │   ├── MediaUploader/
│   │   ├── ContentCalendar/
│   │   └── PostPreview/
│   ├── collaboration/
│   │   ├── CommentSystem/
│   │   ├── ApprovalWorkflow/
│   │   ├── ActivityFeed/
│   │   └── UserManagement/
│   └── analytics/
│       ├── Dashboard/
│       ├── ReportBuilder/
│       └── Charts/
├── hooks/
├── services/
├── stores/
├── utils/
└── types/
```

7.2 State Management with Zustand

typescript

```typescript
// Post store
interface PostStore {
  posts: Post[];
  selectedPost: Post | null;
  isLoading: boolean;
  error: string | null;

  // Actions
  fetchPosts: (workspaceId: string) => Promise<void>;
  createPost: (post: Partial<Post>) => Promise<void>;
  updatePost: (id: string, updates: Partial<Post>) => Promise<void>;
  deletePost: (id: string) => Promise<void>;
  selectPost: (post: Post | null) => void;
}

// Workspace store
interface WorkspaceStore {
  workspaces: Workspace[];
  currentWorkspace: Workspace | null;
  members: WorkspaceMember[];

  // Actions
  fetchWorkspaces: () => Promise<void>;
  switchWorkspace: (workspaceId: string) => void;
  inviteMember: (email: string, role: UserRole) => Promise<void>;
}
```

7.3 Key React Components

Post Composer Component

typescript

```tsx
interface PostComposerProps {
  post?: Post;
  onSave: (post: Partial<Post>) => void;
  onCancel: () => void;
}

const PostComposer: React.FC<PostComposerProps> = ({ post, onSave, onCancel }) => {
  const [content, setContent] = useState(post?.content || {});
  const [selectedPlatforms, setSelectedPlatforms] = useState<SocialPlatform[]>(
    post?.platforms || []
  );
  const [mediaAssets, setMediaAssets] = useState<MediaAsset[]>(
    post?.media_assets || []
  );

  const handleSave = () => {
    onSave({
      content,
      platforms: selectedPlatforms,
      media_assets: mediaAssets,
      status: 'draft'
    });
  };

  return (
    <div className="post-composer">
      <PlatformSelector
        selected={selectedPlatforms}
        onChange={setSelectedPlatforms}
      />
      <ContentEditor
        content={content}
        platforms={selectedPlatforms}
        onChange={setContent}
      />
      <MediaUploader
        assets={mediaAssets}
        onChange={setMediaAssets}
      />
      <div className="actions">
        <Button onClick={handleSave}>Save Draft</Button>
        <Button onClick={onCancel} variant="secondary">Cancel</Button>
      </div>
    </div>
```

```
  );
};
```

Content Calendar Component

typescript

```typescript
const ContentCalendar: React.FC = () => {
  const [view, setView] = useState<'month' | 'week' | 'day'>('month');
  const [selectedDate, setSelectedDate] = useState(new Date());
  const { posts, fetchPosts } = usePostStore();

  const handlePostDrop = (postId: string, newDate: Date) => {
    // Handle drag and drop scheduling
    updatePostSchedule(postId, newDate);
  };

  return (
    <div className="content-calendar">
      <CalendarHeader
        view={view}
        selectedDate={selectedDate}
        onViewChange={setView}
        onDateChange={setSelectedDate}
      />
      <DragDropContext onDragEnd={handlePostDrop}>
        <CalendarGrid
          view={view}
          selectedDate={selectedDate}
          posts={posts}
        />
      </DragDropContext>
    </div>
  );
};
```

8. SOCIAL MEDIA INTEGRATIONS

8.1 Platform API Integrations

Facebook/Instagram Integration

typescript

```typescript
class FacebookService {
  private accessToken: string;

  async publishPost(accountId: string, content: PostContent): Promise<string> {
    const response = await fetch(`https://graph.facebook.com/v18.0/${accountId}/feed`, {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.accessToken}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        message: content.text,
        link: content.link,
        scheduled_publish_time: content.scheduledAt
          ? Math.floor(content.scheduledAt.getTime() / 1000)
          : undefined,
        published: !content.scheduledAt
      })
    });

    const result = await response.json();
    return result.id;
  }

  async getPostAnalytics(postId: string): Promise<PostAnalytics> {
    const response = await fetch(
      `https://graph.facebook.com/v18.0/${postId}/insights?metric=post_impressions,post_engaged_users,p
      {
        headers: {
          'Authorization': `Bearer ${this.accessToken}`
        }
      }
    );

    return response.json();
  }
}
```

Twitter Integration

```typescript
```

```
class TwitterService {
  private client: TwitterApi;

  constructor(accessToken: string, accessSecret: string) {
    this.client = new TwitterApi({
      appKey: process.env.TWITTER_API_KEY!,
      appSecret: process.env.TWITTER_API_SECRET!,
      accessToken,
      accessSecret
    });
  }

  async publishTweet(content: PostContent): Promise<string> {
    const tweetData: any = {
      text: content.text
    };

    if (content.mediaAssets?.length) {
      const mediaIds = await Promise.all(
        content.mediaAssets.map(asset => this.uploadMedia(asset))
      );
      tweetData.media = { media_ids: mediaIds };
    }

    const tweet = await this.client.v2.tweet(tweetData);
    return tweet.data.id;
  }

  private async uploadMedia(asset: MediaAsset): Promise<string> {
    const mediaUpload = await this.client.v1.uploadMedia(asset.url);
    return mediaUpload;
  }
}
```

8.2 Unified Publishing Service

typescript

```
class PublishingService {
  private services: Map<SocialPlatform, any> = new Map();

  constructor() {
    this.services.set(SocialPlatform.FACEBOOK, new FacebookService());
    this.services.set(SocialPlatform.TWITTER, new TwitterService());
    this.services.set(SocialPlatform.LINKEDIN, new LinkedInService());
    // ... other platforms
  }

  async publishToMultiplePlatforms(
    post: Post,
    accounts: SocialAccount[]
  ): Promise<PublishResult[]> {
    const results: PublishResult[] = [];

    for (const account of accounts) {
      try {
        const service = this.services.get(account.platform);
        if (!service) continue;

        const platformContent = post.content.find(c => c.platform === account.platform);
        if (!platformContent) continue;

        const platformPostId = await service.publishPost(account.account_id, platformContent);

        results.push({
          platform: account.platform,
          success: true,
          platformPostId,
          accountId: account.id
        });

        // Update post status
        await this.updatePostStatus(post.id, account.platform, 'published', platformPostId);

      } catch (error) {
        results.push({
          platform: account.platform,
          success: false,
          error: error.message,
          accountId: account.id
        });

        // Update post status
        await this.updatePostStatus(post.id, account.platform, 'failed', null, error.message);
```

```
    }
   }

   return results;
  }
}
```

9. ANALYTICS SYSTEM

9.1 Data Collection Service

typescript

```
class AnalyticsCollectionService {
  private collectors: Map<SocialPlatform, any> = new Map();

  async collectAllMetrics(): Promise<void> {
    const publishedPosts = await this.getPublishedPosts();

    for (const post of publishedPosts) {
      for (const platform of post.platforms) {
        await this.collectPostMetrics(post.id, platform);
      }
    }
  }

  private async collectPostMetrics(postId: string, platform: SocialPlatform): Promise<void> {
    const collector = this.collectors.get(platform);
    if (!collector) return;

    try {
      const metrics = await collector.getPostMetrics(postId);

      await this.storeAnalytics({
        post_id: postId,
        platform,
        metrics,
        collected_at: new Date()
      });

    } catch (error) {
      console.error(`Failed to collect metrics for post ${postId} on ${platform}:`, error);
    }
  }

  private async storeAnalytics(data: AnalyticsData): Promise<void> {
    await db.post_analytics.create({ data });
  }
}
```

9.2 Analytics Dashboard Components

typescript

```tsx
const AnalyticsDashboard: React.FC<{ workspaceId: string }> = ({ workspaceId }) => {
  const [dateRange, setDateRange] = useState({ start: subDays(new Date(), 30), end: new Date() });
  const [metrics, setMetrics] = useState<WorkspaceAnalytics | null>(null);

  useEffect(() => {
    fetchAnalytics(workspaceId, dateRange).then(setMetrics);
  }, [workspaceId, dateRange]);

  if (!metrics) return <LoadingSpinner />;

  return (
    <div className="analytics-dashboard">
      <div className="metrics-overview">
        <MetricCard
          title="Total Reach"
          value={metrics.totalReach}
          change={metrics.reachChange}
        />
        <MetricCard
          title="Engagement Rate"
          value={`${metrics.engagementRate}%`}
          change={metrics.engagementChange}
        />
        <MetricCard
          title="Posts Published"
          value={metrics.postsPublished}
          change={metrics.postsChange}
        />
      </div>

      <div className="charts-section">
        <EngagementChart data={metrics.engagementOverTime} />
        <PlatformPerformance data={metrics.platformMetrics} />
        <TopPerformingPosts posts={metrics.topPosts} />
      </div>

      <div className="detailed-metrics">
        <PostPerformanceTable posts={metrics.allPosts} />
      </div>
    </div>
  );
};
```

10. DEPLOYMENT & INFRASTRUCTURE

10.1 Docker Configuration

dockerfile

```
# Backend Dockerfile
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --only=production

COPY . .
RUN npm run build

EXPOSE 3000

CMD ["npm", "start"]
```

10.2 Docker Compose for Development

yaml

```yaml
version: '3.8'
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: planable_clone
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  backend:
    build: ./backend
    ports:
      - "3000:3000"
    environment:
      DATABASE_URL: postgres://postgres:password@postgres:5432/planable_clone
      REDIS_URL: redis://redis:6379
      JWT_SECRET: your-jwt-secret
    depends_on:
      - postgres
      - redis
    volumes:
      - ./backend:/app
      - /app/node_modules

  frontend:
    build: ./frontend
    ports:
      - "3001:3000"
    environment:
      REACT_APP_API_URL: http://localhost:3000
    volumes:
      - ./frontend:/app
      - /app/node_modules

volumes:
  postgres_data:
```

10.3 Production Deployment (AWS)

yaml

```yaml
# kubernetes/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: planable-clone-backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: planable-clone-backend
  template:
    metadata:
      labels:
        app: planable-clone-backend
    spec:
      containers:
      - name: backend
        image: your-registry/planable-clone-backend:latest
        ports:
        - containerPort: 3000
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: app-secrets
              key: database-url
        - name: REDIS_URL
          valueFrom:
            secretKeyRef:
              name: app-secrets
              key: redis-url
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"

---
apiVersion: v1
kind: Service
metadata:
  name: planable-clone-backend-service
spec:
  selector:
```

```
  app: planable-clone-backend
ports:
- protocol: TCP
  port: 80
  targetPort: 3000
type: LoadBalancer
```

10.4 Infrastructure as Code (Terraform)

```hcl
```

```
# infrastructure/main.tf
provider "aws" {
  region = var.aws_region
}

# VPC and Networking
resource "aws_vpc" "main" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support   = true

  tags = {
    Name = "planable-clone-vpc"
  }
}

resource "aws_subnet" "public" {
  count             = 2
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.${count.index + 1}.0/24"
  availability_zone = data.aws_availability_zones.available.names[count.index]

  map_public_ip_on_launch = true

  tags = {
    Name = "planable-clone-public-${count.index + 1}"
  }
}

resource "aws_subnet" "private" {
  count             = 2
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.${count.index + 10}.0/24"
  availability_zone = data.aws_availability_zones.available.names[count.index]

  tags = {
    Name = "planable-clone-private-${count.index + 1}"
  }
}

# RDS Database
resource "aws_db_subnet_group" "main" {
  name       = "planable-clone-db-subnet-group"
  subnet_ids = aws_subnet.private[*].id

  tags = {
```

```
    Name = "planable-clone-db-subnet-group"
  }
}

resource "aws_db_instance" "postgres" {
  identifier     = "planable-clone-db"
  engine         = "postgres"
  engine_version = "15.3"
  instance_class = "db.t3.micro"

  allocated_storage     = 20
  max_allocated_storage = 100
  storage_type          = "gp2"
  storage_encrypted     = true

  db_name  = "planable_clone"
  username = var.db_username
  password = var.db_password

  vpc_security_group_ids = [aws_security_group.rds.id]
  db_subnet_group_name   = aws_db_subnet_group.main.name

  backup_retention_period = 7
  backup_window           = "03:00-04:00"
  maintenance_window      = "sun:04:00-sun:05:00"

  skip_final_snapshot = false
  final_snapshot_identifier = "planable-clone-final-snapshot"

  tags = {
    Name = "planable-clone-db"
  }
}

# ElastiCache Redis
resource "aws_elasticache_subnet_group" "main" {
  name       = "planable-clone-cache-subnet"
  subnet_ids = aws_subnet.private[*].id
}

resource "aws_elasticache_cluster" "redis" {
  cluster_id           = "planable-clone-redis"
  engine               = "redis"
  node_type            = "cache.t3.micro"
  num_cache_nodes      = 1
  parameter_group_name = "default.redis7"
  port                 = 6379
```

```
  subnet_group_name  = aws_elasticache_subnet_group.main.name
  security_group_ids = [aws_security_group.redis.id]

  tags = {
    Name = "planable-clone-redis"
  }
}

# ECS Cluster
resource "aws_ecs_cluster" "main" {
  name = "planable-clone-cluster"

  setting {
    name  = "containerInsights"
    value = "enabled"
  }

  tags = {
    Name = "planable-clone-cluster"
  }
}

# Application Load Balancer
resource "aws_lb" "main" {
  name               = "planable-clone-alb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.alb.id]
  subnets            = aws_subnet.public[*].id

  enable_deletion_protection = false

  tags = {
    Name = "planable-clone-alb"
  }
}

# S3 Bucket for Media Storage
resource "aws_s3_bucket" "media" {
  bucket = "planable-clone-media-${random_string.bucket_suffix.result}"

  tags = {
    Name = "planable-clone-media"
  }
}

resource "aws_s3_bucket_cors_configuration" "media" {
```

```
  bucket = aws_s3_bucket.media.id

  cors_rule {
    allowed_headers = ["*"]
    allowed_methods = ["GET", "PUT", "POST", "DELETE", "HEAD"]
    allowed_origins = ["*"]
    expose_headers  = ["ETag"]
    max_age_seconds = 3000
  }
}

resource "random_string" "bucket_suffix" {
  length  = 8
  special = false
  upper   = false
}
```

11. SECURITY IMPLEMENTATION

11.1 Authentication & Authorization

typescript

```typescript
// JWT middleware with refresh token rotation
class AuthService {
  private readonly accessTokenExpiry = '15m';
  private readonly refreshTokenExpiry = '7d';

  async generateTokenPair(userId: string): Promise<TokenPair> {
    const payload = { userId, type: 'access' };

    const accessToken = jwt.sign(payload, process.env.JWT_SECRET!, {
      expiresIn: this.accessTokenExpiry
    });

    const refreshToken = jwt.sign(
      { userId, type: 'refresh' },
      process.env.JWT_REFRESH_SECRET!,
      { expiresIn: this.refreshTokenExpiry }
    );

    // Store refresh token hash in database
    await this.storeRefreshToken(userId, refreshToken);

    return { accessToken, refreshToken };
  }

  async refreshTokens(refreshToken: string): Promise<TokenPair> {
    try {
      const decoded = jwt.verify(refreshToken, process.env.JWT_REFRESH_SECRET!) as any;

      // Verify refresh token exists in database
      const storedToken = await this.getStoredRefreshToken(decoded.userId);
      if (!storedToken || !await bcrypt.compare(refreshToken, storedToken.hash)) {
        throw new Error('Invalid refresh token');
      }

      // Generate new token pair
      const newTokens = await this.generateTokenPair(decoded.userId);

      // Invalidate old refresh token
      await this.invalidateRefreshToken(decoded.userId, refreshToken);

      return newTokens;
    } catch (error) {
      throw new Error('Invalid refresh token');
    }
  }
}
```

```typescript
// Permission-based middleware
const requirePermission = (permission: Permission) => {
  return async (req: AuthenticatedRequest, res: Response, next: NextFunction) => {
    const user = req.user;
    const workspaceId = req.params.workspaceId;

    const hasPermission = await checkUserPermission(user.id, workspaceId, permission);

    if (!hasPermission) {
      return res.status(403).json({ error: 'Insufficient permissions' });
    }

    next();
  };
};

// Usage in routes
router.put('/workspaces/:workspaceId/posts/:postId',
  authenticateToken,
  requirePermission(Permission.EDIT_POSTS),
  updatePost
);
```

11.2 Rate Limiting & Security Headers

typescript

```
// Rate limiting configuration
const rateLimitConfig = {
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP, please try again later.',
  standardHeaders: true,
  legacyHeaders: false,
};

// Security middleware setup
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'", "https://fonts.googleapis.com"],
      fontSrc: ["'self'", "https://fonts.gstatic.com"],
      imgSrc: ["'self'", "data:", "https:"],
      scriptSrc: ["'self'"],
      connectSrc: ["'self'", "https://api.planable-clone.com"],
    },
  },
  hsts: {
    maxAge: 31536000,
    includeSubDomains: true,
    preload: true
  }
}));

app.use(rateLimit(rateLimitConfig));
app.use(compression());
app.use(cors({
  origin: process.env.FRONTEND_URL,
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH'],
  allowedHeaders: ['Content-Type', 'Authorization']
}));
```

11.3 Data Encryption & Privacy

typescript

```
// Encryption service for sensitive data
class EncryptionService {
  private readonly algorithm = 'aes-256-gcm';
  private readonly key = crypto.scryptSync(process.env.ENCRYPTION_KEY!, 'salt', 32);

  encrypt(text: string): EncryptedData {
    const iv = crypto.randomBytes(16);
    const cipher = crypto.createCipher(this.algorithm, this.key);
    cipher.setAAD(Buffer.from('planable-clone'));

    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');

    const authTag = cipher.getAuthTag();

    return {
      encrypted,
      iv: iv.toString('hex'),
      authTag: authTag.toString('hex')
    };
  }

  decrypt(encryptedData: EncryptedData): string {
    const decipher = crypto.createDecipher(this.algorithm, this.key);
    decipher.setAAD(Buffer.from('planable-clone'));
    decipher.setAuthTag(Buffer.from(encryptedData.authTag, 'hex'));

    let decrypted = decipher.update(encryptedData.encrypted, 'hex', 'utf8');
    decrypted += decipher.final('utf8');

    return decrypted;
  }
}

// Social media token encryption
class SocialAccountService {
  private encryptionService = new EncryptionService();

  async storeSocialAccount(accountData: SocialAccountData): Promise<void> {
    const encryptedToken = this.encryptionService.encrypt(accountData.accessToken);
    const encryptedRefreshToken = accountData.refreshToken
      ? this.encryptionService.encrypt(accountData.refreshToken)
      : null;

    await db.social_accounts.create({
      data: {
```

```
      ...accountData,
      access_token: JSON.stringify(encryptedToken),
      refresh_token: encryptedRefreshToken ? JSON.stringify(encryptedRefreshToken) : null
    }
   });
  }
}
```

12. TESTING STRATEGY

12.1 Unit Testing Setup

typescript

```
// Jest configuration
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  roots: ['<rootDir>/src'],
  testMatch: ['**/__tests__/**/*.test.ts'],
  collectCoverageFrom: [
    'src/**/*.{ts,tsx}',
    '!src/**/*.d.ts',
    '!src/index.ts'
  ],
  coverageReporters: ['text', 'lcov', 'html'],
  setupFilesAfterEnv: ['<rootDir>/src/test/setup.ts']
};

// Example unit tests
describe('PostService', () => {
  let postService: PostService;
  let mockDb: jest.Mocked<Database>;

  beforeEach(() => {
    mockDb = createMockDatabase();
    postService = new PostService(mockDb);
  });

  describe('createPost', () => {
    it('should create a post with valid data', async () => {
      const postData = {
        title: 'Test Post',
        content: [{ platform: 'facebook', text: 'Hello World' }],
        workspaceId: 'workspace-1'
      };

      mockDb.posts.create.mockResolvedValue({ id: 'post-1', ...postData });

      const result = await postService.createPost(postData);

      expect(result.id).toBe('post-1');
      expect(mockDb.posts.create).toHaveBeenCalledWith({
        data: expect.objectContaining(postData)
      });
    });

    it('should throw error for invalid platform content', async () => {
      const postData = {
        title: 'Test Post',
```

```
      content: [{ platform: 'facebook', text: '' }], // Empty text
      workspaceId: 'workspace-1'
    };

    await expect(postService.createPost(postData)).rejects.toThrow(
      'Content cannot be empty for platform facebook'
    );
  });
 });
});

// Integration tests
describe('Post API Endpoints', () => {
 let app: Express;
 let testDb: Database;

 beforeAll(async () => {
  testDb = await createTestDatabase();
  app = createApp(testDb);
 });

 afterAll(async () => {
  await cleanupTestDatabase(testDb);
 });

 describe('POST /api/workspaces/:workspaceId/posts', () => {
  it('should create a new post', async () => {
   const user = await createTestUser();
   const workspace = await createTestWorkspace(user.id);
   const token = generateTestToken(user.id);

   const response = await request(app)
    .post(`/api/workspaces/${workspace.id}/posts`)
    .set('Authorization', `Bearer ${token}`)
    .send({
     title: 'Test Post',
     content: [{ platform: 'facebook', text: 'Hello World' }],
     platforms: ['facebook']
    });

   expect(response.status).toBe(201);
   expect(response.body.title).toBe('Test Post');
  });
 });
});
```

12.2 End-to-End Testing with Playwright

typescript

```typescript
// e2e/tests/post-creation.spec.ts
import { test, expect } from '@playwright/test';

test.describe('Post Creation Flow', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('/login');
    await page.fill('[data-testid=email]', 'test@example.com');
    await page.fill('[data-testid=password]', 'password123');
    await page.click('[data-testid=login-button]');
    await page.waitForURL('/dashboard');
  });

  test('should create a new post successfully', async ({ page }) => {
    // Navigate to post creation
    await page.click('[data-testid=create-post-button]');
    await page.waitForSelector('[data-testid=post-composer]');

    // Fill post content
    await page.fill('[data-testid=post-title]', 'My Test Post');
    await page.fill('[data-testid=post-content]', 'This is a test post content');

    // Select platforms
    await page.check('[data-testid=platform-facebook]');
    await page.check('[data-testid=platform-twitter]');

    // Upload media
    await page.setInputFiles('[data-testid=media-upload]', 'test-image.jpg');
    await page.waitForSelector('[data-testid=uploaded-media]');

    // Save post
    await page.click('[data-testid=save-post-button]');

    // Verify post was created
    await expect(page.locator('[data-testid=success-message]')).toContainText('Post saved successfully');
    await page.waitForURL('/dashboard');

    // Verify post appears in calendar
    await expect(page.locator('[data-testid=calendar-post]').first()).toContainText('My Test Post');
  });

  test('should handle approval workflow', async ({ page }) => {
    // Create post that requires approval
    await page.click('[data-testid=create-post-button]');
    await page.fill('[data-testid=post-title]', 'Post Requiring Approval');
    await page.fill('[data-testid=post-content]', 'This post needs approval');
    await page.check('[data-testid=platform-facebook]');
```

```
    // Submit for approval
    await page.click('[data-testid=submit-for-approval-button]');

    // Verify approval status
    await expect(page.locator('[data-testid=post-status]')).toContainText('Pending Approval');

    // Switch to approver account (simulate)
    await page.goto('/logout');
    await page.goto('/login');
    await page.fill('[data-testid=email]', 'approver@example.com');
    await page.fill('[data-testid=password]', 'password123');
    await page.click('[data-testid=login-button]');

    // Approve the post
    await page.goto('/approvals');
    await page.click('[data-testid=approve-button]');
    await page.fill('[data-testid=approval-comment]', 'Looks good!');
    await page.click('[data-testid=confirm-approval-button]');

    // Verify approval
    await expect(page.locator('[data-testid=post-status]')).toContainText('Approved');
  });
});
```

13. MONITORING & OBSERVABILITY

13.1 Application Monitoring

typescript

```
// Monitoring service with Prometheus metrics
import prometheus from 'prom-client';

class MonitoringService {
  private httpRequestDuration = new prometheus.Histogram({
    name: 'http_request_duration_seconds',
    help: 'Duration of HTTP requests in seconds',
    labelNames: ['method', 'route', 'status'],
    buckets: [0.1, 0.5, 1, 2, 5]
  });

  private postsPublished = new prometheus.Counter({
    name: 'posts_published_total',
    help: 'Total number of posts published',
    labelNames: ['platform', 'workspace']
  });

  private socialApiErrors = new prometheus.Counter({
    name: 'social_api_errors_total',
    help: 'Total number of social media API errors',
    labelNames: ['platform', 'error_type']
  });

  trackHttpRequest(method: string, route: string, status: number, duration: number) {
    this.httpRequestDuration
      .labels(method, route, status.toString())
      .observe(duration);
  }

  trackPostPublished(platform: string, workspaceId: string) {
    this.postsPublished.labels(platform, workspaceId).inc();
  }

  trackSocialApiError(platform: string, errorType: string) {
    this.socialApiErrors.labels(platform, errorType).inc();
  }

  getMetrics() {
    return prometheus.register.metrics();
  }
}

// Express middleware for request tracking
const requestTracker = (monitoring: MonitoringService) => {
  return (req: Request, res: Response, next: NextFunction) => {
    const start = Date.now();
```

```
    res.on('finish', () => {
      const duration = (Date.now() - start) / 1000;
      monitoring.trackHttpRequest(req.method, req.route?.path || req.url, res.statusCode, duration);
    });

    next();
  };
};
```

13.2 Logging Strategy

typescript

```typescript
// Structured logging with Winston
import winston from 'winston';

const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  defaultMeta: { service: 'planable-clone' },
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
    new winston.transports.Console({
      format: winston.format.combine(
        winston.format.colorize(),
        winston.format.simple()
      )
    })
  ]
});

// Usage throughout the application
class PostService {
  async publishPost(postId: string, platforms: string[]): Promise<void> {
    logger.info('Starting post publication', {
      postId,
      platforms,
      userId: this.getCurrentUserId()
    });

    try {
      for (const platform of platforms) {
        await this.publishToPlatform(postId, platform);
        logger.info('Post published successfully', { postId, platform });
      }
    } catch (error) {
      logger.error('Post publication failed', {
        postId,
        platforms,
        error: error.message,
        stack: error.stack
      });
      throw error;
    }
```

```
  }
}
```

13.3 Health Check Endpoints

typescript

```
// Health check service
class HealthCheckService {
  async checkDatabase(): Promise<HealthStatus> {
    try {
      await db.$queryRaw`SELECT 1`;
      return { status: 'healthy', message: 'Database connection successful' };
    } catch (error) {
      return { status: 'unhealthy', message: `Database error: ${error.message}` };
    }
  }

  async checkRedis(): Promise<HealthStatus> {
    try {
      await redis.ping();
      return { status: 'healthy', message: 'Redis connection successful' };
    } catch (error) {
      return { status: 'unhealthy', message: `Redis error: ${error.message}` };
    }
  }

  async checkSocialMediaAPIs(): Promise<Record<string, HealthStatus>> {
    const results: Record<string, HealthStatus> = {};

    const platforms = ['facebook', 'twitter', 'linkedin'];

    await Promise.all(platforms.map(async (platform) => {
      try {
        await this.testPlatformAPI(platform);
        results[platform] = { status: 'healthy', message: 'API accessible' };
      } catch (error) {
        results[platform] = { status: 'unhealthy', message: error.message };
      }
    }));

    return results;
  }

  async getOverallHealth(): Promise<SystemHealth> {
    const [database, redis, socialAPIs] = await Promise.all([
      this.checkDatabase(),
      this.checkRedis(),
      this.checkSocialMediaAPIs()
    ]);

    const isHealthy = database.status === 'healthy' &&
                      redis.status === 'healthy' &&
```

```
                Object.values(socialAPIs).every(api => api.status === 'healthy');

    return {
      status: isHealthy ? 'healthy' : 'unhealthy',
      timestamp: new Date().toISOString(),
      services: {
        database,
        redis,
        socialAPIs
      }
    };
  }
}

// Health check endpoints
router.get('/health', async (req, res) => {
  const health = await healthCheckService.getOverallHealth();
  const statusCode = health.status === 'healthy' ? 200 : 503;
  res.status(statusCode).json(health);
});

router.get('/health/ready', async (req, res) => {
  // Readiness probe for Kubernetes
  const isReady = await checkApplicationReadiness();
  res.status(isReady ? 200 : 503).json({ ready: isReady });
});

router.get('/health/live', (req, res) => {
  // Liveness probe for Kubernetes
  res.status(200).json({ alive: true });
});
```

14. PERFORMANCE OPTIMIZATION

14.1 Database Optimization

```sql
```

```sql
-- Database performance optimizations
-- Partitioning for analytics data
CREATE TABLE post_analytics_2024 PARTITION OF post_analytics
FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE post_analytics_2025 PARTITION OF post_analytics
FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');

-- Materialized views for common queries
CREATE MATERIALIZED VIEW workspace_analytics_summary AS
SELECT
  w.id as workspace_id,
  w.name as workspace_name,
  COUNT(p.id) as total_posts,
  COUNT(CASE WHEN p.status = 'published' THEN 1 END) as published_posts,
  AVG(CASE WHEN pa.metrics->>'engagement_rate' IS NOT NULL
      THEN (pa.metrics->>'engagement_rate')::float END) as avg_engagement_rate,
  SUM(CASE WHEN pa.metrics->>'reach' IS NOT NULL
      THEN (pa.metrics->>'reach')::int END) as total_reach
FROM workspaces w
LEFT JOIN posts p ON w.id = p.workspace_id
LEFT JOIN post_analytics pa ON p.id = pa.post_id
WHERE p.created_at >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY w.id, w.name;

-- Refresh materialized view periodically
CREATE OR REPLACE FUNCTION refresh_analytics_summary()
RETURNS void AS $
BEGIN
  REFRESH MATERIALIZED VIEW CONCURRENTLY workspace_analytics_summary;
END;
$ LANGUAGE plpgsql;

-- Connection pooling configuration
-- Set in postgresql.conf
max_connections = 100
shared_buffers = 256MB
effective_cache_size = 1GB
work_mem = 4MB
maintenance_work_mem = 64MB
```

14.2 Caching Strategy

typescript

```typescript
// Multi-level caching implementation
class CacheService {
  private redis: Redis;
  private localCache: NodeCache;

  constructor() {
    this.redis = new Redis(process.env.REDIS_URL);
    this.localCache = new NodeCache({ stdTTL: 300 }); // 5 minutes
  }

  async get<T>(key: string): Promise<T | null> {
    // Try local cache first
    const localValue = this.localCache.get<T>(key);
    if (localValue !== undefined) {
      return localValue;
    }

    // Try Redis cache
    const redisValue = await this.redis.get(key);
    if (redisValue) {
      const parsed = JSON.parse(redisValue);
      this.localCache.set(key, parsed); // Store in local cache
      return parsed;
    }

    return null;
  }

  async set(key: string, value: any, ttl: number = 3600): Promise<void> {
    const serialized = JSON.stringify(value);

    // Set in both caches
    await this.redis.setex(key, ttl, serialized);
    this.localCache.set(key, value, ttl);
  }

  async invalidate(pattern: string): Promise<void> {
    // Invalidate Redis keys
    const keys = await this.redis.keys(pattern);
    if (keys.length > 0) {
      await this.redis.del(...keys);
    }

    // Invalidate local cache
    this.localCache.flushAll();
  }
```

```
}

// Cache-aside pattern for common queries
class PostService {
  constructor(private cache: CacheService) {}

  async getWorkspacePosts(workspaceId: string, page: number = 1): Promise<Post[]> {
    const cacheKey = `workspace:${workspaceId}:posts:page:${page}`;

    // Try cache first
    let posts = await this.cache.get<Post[]>(cacheKey);
    if (posts) {
      return posts;
    }

    // Fetch from database
    posts = await db.posts.findMany({
      where: { workspace_id: workspaceId },
      skip: (page - 1) * 20,
      take: 20,
      orderBy: { created_at: 'desc' }
    });

    // Cache the result
    await this.cache.set(cacheKey, posts, 900); // 15 minutes

    return posts;
  }

  async updatePost(postId: string, updates: Partial<Post>): Promise<Post> {
    const post = await db.posts.update({
      where: { id: postId },
      data: updates
    });

    // Invalidate related caches
    await this.cache.invalidate(`workspace:${post.workspace_id}:posts:*`);
    await this.cache.invalidate(`post:${postId}:*`);

    return post;
  }
}
```

14.3 Frontend Performance

typescript

```tsx
// React performance optimizations

// 1. Code splitting and lazy loading
const PostComposer = lazy(() => import('./components/PostComposer'));
const AnalyticsDashboard = lazy(() => import('./components/AnalyticsDashboard'));

const App: React.FC = () => {
  return (
    <Router>
      <Suspense fallback={<LoadingSpinner />}>
        <Routes>
          <Route path="/compose" element={<PostComposer />} />
          <Route path="/analytics" element={<AnalyticsDashboard />} />
        </Routes>
      </Suspense>
    </Router>
  );
};

// 2. Memoization for expensive components
const PostCard = React.memo<PostCardProps>(({ post, onUpdate }) => {
  const handleUpdate = useCallback((updates: Partial<Post>) => {
    onUpdate(post.id, updates);
  }, [post.id, onUpdate]);

  return (
    <div className="post-card">
      <h3>{post.title}</h3>
      <p>{post.content[0]?.text}</p>
      <PostActions post={post} onUpdate={handleUpdate} />
    </div>
  );
});

// 3. Virtual scrolling for large lists
const VirtualizedPostList: React.FC<{ posts: Post[] }> = ({ posts }) => {
  const parentRef = useRef<HTMLDivElement>(null);

  const rowVirtualizer = useVirtualizer({
    count: posts.length,
    getScrollElement: () => parentRef.current,
    estimateSize: () => 200,
    overscan: 5,
  });

  return (
```

```
    <div ref={parentRef} className="virtual-list-container">
      <div
        style={{
          height: `${rowVirtualizer.getTotalSize()}px`,
          width: '100%',
          position: 'relative',
        }}
      >
        {rowVirtualizer.getVirtualItems().map((virtualItem) => (
          <div
            key={virtualItem.key}
            style={{
              position: 'absolute',
              top: 0,
              left: 0,
              width: '100%',
              height: `${virtualItem.size}px`,
              transform: `translateY(${virtualItem.start}px)`,
            }}
          >
            <PostCard post={posts[virtualItem.index]} />
          </div>
        ))}
      </div>
    </div>
  );
};

// 4. Optimistic updates
const useOptimisticPosts = () => {
  const [posts, setPosts] = useState<Post[]>([]);
  const [optimisticUpdates, setOptimisticUpdates] = useState<Map<string, Partial<Post>>>(new Map());

  const updatePostOptimistically = useCallback(async (postId: string, updates: Partial<Post>) => {
    // Apply optimistic update immediately
    setOptimisticUpdates(prev => new Map(prev).set(postId, updates));

    try {
      // Make actual API call
      const updatedPost = await updatePost(postId, updates);

      // Update actual state and clear optimistic update
      setPosts(prev => prev.map(p => p.id === postId ? updatedPost : p));
      setOptimisticUpdates(prev => {
        const newMap = new Map(prev);
        newMap.delete(postId);
        return newMap;
```

```
      });
    } catch (error) {
      // Revert optimistic update on error
      setOptimisticUpdates(prev => {
        const newMap = new Map(prev);
        newMap.delete(postId);
        return newMap;
      });
      throw error;
    }
  }, []);

  // Merge optimistic updates with actual posts
  const displayPosts = useMemo(() => {
    return posts.map(post => {
      const optimisticUpdate = optimisticUpdates.get(post.id);
      return optimisticUpdate ? { ...post, ...optimisticUpdate } : post;
    });
  }, [posts, optimisticUpdates]);

  return { posts: displayPosts, updatePostOptimistically };
};
```

15. SCALABILITY ARCHITECTURE

15.1 Microservices Decomposition

typescript

```
// Service registry and discovery
class ServiceRegistry {
  private services: Map<string, ServiceInstance[]> = new Map();

  registerService(serviceName: string, instance: ServiceInstance): void {
    const instances = this.services.get(serviceName) || [];
    instances.push(instance);
    this.services.set(serviceName, instances);
  }

  discoverService(serviceName: string): ServiceInstance | null {
    const instances = this.services.get(serviceName) || [];
    if (instances.length === 0) return null;

    // Simple round-robin load balancing
    const index = Math.floor(Math.random() * instances.length);
    return instances[index];
  }

  healthCheck(): void {
    // Periodically check service health
    setInterval(async () => {
      for (const [serviceName, instances] of this.services.entries()) {
        const healthyInstances = await Promise.all(
          instances.map(async (instance) => {
            try {
              const response = await fetch(`${instance.url}/health`);
              return response.ok ? instance : null;
            } catch {
              return null;
            }
          })
        );

        this.services.set(serviceName, healthyInstances.filter(Boolean) as ServiceInstance[]);
      }
    }, 30000); // Check every 30 seconds
  }
}

// API Gateway with service routing
class APIGateway {
  constructor(private serviceRegistry: ServiceRegistry) {}

  async routeRequest(req: Request, res: Response): Promise<void> {
    const serviceName = this.extractServiceName(req.path);
```

```
    const serviceInstance = this.serviceRegistry.discoverService(serviceName);

    if (!serviceInstance) {
      res.status(503).json({ error: 'Service unavailable' });
      return;
    }

    try {
      const response = await this.forwardRequest(req, serviceInstance);
      res.status(response.status).json(response.data);
    } catch (error) {
      res.status(500).json({ error: 'Internal server error' });
    }
  }

  private extractServiceName(path: string): string {
    // Extract service name from path: /api/posts/* -> posts-service
    const segments = path.split('/');
    return segments[2] ? `${segments[2]}-service` : 'unknown';
  }

  private async forwardRequest(req: Request, instance: ServiceInstance): Promise<any> {
    const url = `${instance.url}${req.path}`;
    const response = await fetch(url, {
      method: req.method,
      headers: req.headers as any,
      body: req.method !== 'GET' ? JSON.stringify(req.body) : undefined
    });

    return {
      status: response.status,
      data: await response.json()
    };
  }
}
```

15.2 Event-Driven Architecture

typescript

```typescript
// Event bus implementation
interface DomainEvent {
  id: string;
  type: string;
  aggregateId: string;
  aggregateType: string;
  data: any;
  timestamp: Date;
  version: number;
}

class EventBus {
  private handlers: Map<string, EventHandler[]> = new Map();
  private eventStore: EventStore;

  constructor(eventStore: EventStore) {
    this.eventStore = eventStore;
  }

  async publish(event: DomainEvent): Promise<void> {
    // Store event
    await this.eventStore.save(event);

    // Publish to handlers
    const handlers = this.handlers.get(event.type) || [];
    await Promise.all(handlers.map(handler => handler.handle(event)));

    // Publish to message queue for cross-service communication
    await this.publishToQueue(event);
  }

  subscribe(eventType: string, handler: EventHandler): void {
    const handlers = this.handlers.get(eventType) || [];
    handlers.push(handler);
    this.handlers.set(eventType, handlers);
  }

  private async publishToQueue(event: DomainEvent): Promise<void> {
    // Publish to Redis Streams or RabbitMQ
    await redis.xadd('events', '*', 'data', JSON.stringify(event));
  }
}

// Event handlers
class PostPublishedHandler implements EventHandler {
  async handle(event: DomainEvent): Promise<void> {
```

```
    if (event.type === 'PostPublished') {
      // Update analytics
      await this.updateAnalytics(event.data);

      // Send notifications
      await this.sendNotifications(event.data);

      // Update search index
      await this.updateSearchIndex(event.data);
    }
  }

  private async updateAnalytics(postData: any): Promise<void> {
    // Increment published posts counter
    await redis.incr(`analytics:workspace:${postData.workspaceId}:posts_published`);
  }

  private async sendNotifications(postData: any): Promise<void> {
    // Send notifications to workspace members
    const members = await this.getWorkspaceMembers(postData.workspaceId);

    for (const member of members) {
      await this.notificationService.send({
        userId: member.id,
        type: 'post_published',
        title: 'Post Published',
        message: `"${postData.title}" has been published successfully`,
        data: { postId: postData.id }
      });
    }
  }
}

// Saga pattern for complex workflows
class PostApprovalSaga {
  constructor(private eventBus: EventBus) {
    this.eventBus.subscribe('PostSubmittedForApproval', this);
    this.eventBus.subscribe('PostApproved', this);
    this.eventBus.subscribe('PostRejected', this);
  }

  async handle(event: DomainEvent): Promise<void> {
    switch (event.type) {
      case 'PostSubmittedForApproval':
        await this.handleSubmissionForApproval(event);
        break;
      case 'PostApproved':
```

```
      await this.handleApproval(event);
      break;
    case 'PostRejected':
      await this.handleRejection(event);
      break;
  }
}

private async handleSubmissionForApproval(event: DomainEvent): Promise<void> {
  const { postId, workspaceId } = event.data;

  // Get approval workflow
  const workflow = await this.getApprovalWorkflow(workspaceId);

  // Start approval process
  await this.eventBus.publish({
    id: uuid(),
    type: 'ApprovalProcessStarted',
    aggregateId: postId,
    aggregateType: 'Post',
    data: {
      postId,
      workflowId: workflow.id,
      currentStep: 0,
      approvers: workflow.steps[0].approvers
    },
    timestamp: new Date(),
    version: 1
  });

  // Send notifications to approvers
  await this.notifyApprovers(workflow.steps[0].approvers, postId);
}

private async handleApproval(event: DomainEvent): Promise<void> {
  const { postId, approvalId, stepIndex } = event.data;

  const workflow = await this.getPostApprovalWorkflow(postId);
  const nextStep = stepIndex + 1;

  if (nextStep < workflow.steps.length) {
    // Move to next approval step
    await this.eventBus.publish({
      id: uuid(),
      type: 'NextApprovalStepStarted',
      aggregateId: postId,
      aggregateType: 'Post',
```

```
        data: {
          postId,
          currentStep: nextStep,
          approvers: workflow.steps[nextStep].approvers
        },
        timestamp: new Date(),
        version: 1
      });
    } else {
      // All approvals complete - publish post
      await this.eventBus.publish({
        id: uuid(),
        type: 'PostReadyForPublishing',
        aggregateId: postId,
        aggregateType: 'Post',
        data: { postId },
        timestamp: new Date(),
        version: 1
      });
    }
  }
}
```

15.3 Database Sharding Strategy

typescript

```
// Database sharding implementation
class ShardManager {
  private shards: DatabaseShard[];

  constructor(shards: DatabaseShard[]) {
    this.shards = shards;
  }

  getShardForWorkspace(workspaceId: string): DatabaseShard {
    // Consistent hashing for workspace distribution
    const hash = this.hash(workspaceId);
    const shardIndex = hash % this.shards.length;
    return this.shards[shardIndex];
  }

  getShardForUser(userId: string): DatabaseShard {
    const hash = this.hash(userId);
    const shardIndex = hash % this.shards.length;
    return this.shards[shardIndex];
  }

  private hash(input: string): number {
    let hash = 0;
    for (let i = 0; i < input.length; i++) {
      const char = input.charCodeAt(i);
      hash = ((hash << 5) - hash) + char;
      hash = hash & hash; // Convert to 32-bit integer
    }
    return Math.abs(hash);
  }

  async executeQuery(workspaceId: string, query: string, params: any[]): Promise<any> {
    const shard = this.getShardForWorkspace(workspaceId);
    return shard.query(query, params);
  }

  async executeQueryOnAllShards(query: string, params: any[]): Promise<any[]> {
    const results = await Promise.all(
      this.shards.map(shard => shard.query(query, params))
    );
    return results.flat();
  }
}

// Repository pattern with sharding
class PostRepository {
```

```
constructor(private shardManager: ShardManager) {}

async create(post: CreatePostData): Promise<Post> {
  const shard = this.shardManager.getShardForWorkspace(post.workspaceId);

  return shard.posts.create({
    data: {
      ...post,
      id: uuid(),
      created_at: new Date()
    }
  });
}

async findByWorkspace(workspaceId: string, options: FindOptions): Promise<Post[]> {
  const shard = this.shardManager.getShardForWorkspace(workspaceId);

  return shard.posts.findMany({
    where: { workspace_id: workspaceId },
    ...options
  });
}

async findById(postId: string, workspaceId: string): Promise<Post | null> {
  const shard = this.shardManager.getShardForWorkspace(workspaceId);

  return shard.posts.findUnique({
    where: { id: postId }
  });
}

async update(postId: string, workspaceId: string, updates: Partial<Post>): Promise<Post> {
  const shard = this.shardManager.getShardForWorkspace(workspaceId);

  return shard.posts.update({
    where: { id: postId },
    data: updates
  });
}

// Cross-shard queries (for admin/analytics)
async findAllPosts(criteria: SearchCriteria): Promise<Post[]> {
  const results = await this.shardManager.executeQueryOnAllShards(
    'SELECT * FROM posts WHERE created_at >= $1 AND created_at <= $2',
    [criteria.startDate, criteria.endDate]
  );
```

```
    return results.flat();
  }
}
```

16. MOBILE CONSIDERATIONS

16.1 Progressive Web App (PWA)

typescript

```
// Service Worker for offline functionality
// sw.js
const CACHE_NAME = 'planable-clone-v1';
const urlsToCache = [
  '/',
  '/static/js/bundle.js',
  '/static/css/main.css',
  '/manifest.json'
];

self.addEventListener('install', (event: ExtendableEvent) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => cache.addAll(urlsToCache))
  );
});

self.addEventListener('fetch', (event: FetchEvent) => {
  event.respondWith(
    caches.match(event.request)
      .then((response) => {
        // Return cached version or fetch from network
        return response || fetch(event.request);
      })
  );
});

// Push notifications
self.addEventListener('push', (event: PushEvent) => {
  const options = {
    body: event.data?.text() || 'New notification',
    icon: '/icon-192x192.png',
    badge: '/badge-72x72.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: '2'
    },
    actions: [
      {
        action: 'explore',
        title: 'View',
        icon: '/images/checkmark.png'
      },
      {
        action: 'close',
```

```
      title: 'Close',
      icon: '/images/xmark.png'
    }
  ]
 };

 event.waitUntil(
  self.registration.showNotification('Planable Clone', options)
 );
});

// Web App Manifest
// manifest.json
{
 "name": "Planable Clone",
 "short_name": "Planable",
 "description": "Social Media Management Platform",
 "start_url": "/",
 "display": "standalone",
 "background_color": "#ffffff",
 "theme_color": "#4f46e5",
 "icons": [
  {
   "src": "/icon-192x192.png",
   "sizes": "192x192",
   "type": "image/png"
  },
  {
   "src": "/icon-512x512.png",
   "sizes": "512x512",
   "type": "image/png"
  }
 ],
 "categories": ["productivity", "social"],
 "shortcuts": [
  {
   "name": "Create Post",
   "short_name": "New Post",
   "description": "Create a new social media post",
   "url": "/compose",
   "icons": [{ "src": "/icon-96x96.png", "sizes": "96x96" }]
  },
  {
   "name": "Calendar",
   "short_name": "Calendar",
   "description": "View content calendar",
   "url": "/calendar",
```

```
    "icons": [{ "src": "/icon-96x96.png", "sizes": "96x96" }]
   }
  ]
}
```

16.2 Mobile-Responsive Components

typescript

```tsx
// Mobile-optimized calendar component
const MobileCalendar: React.FC<CalendarProps> = ({ posts, onPostSelect }) => {
  const [view, setView] = useState<'agenda' | 'month'>('agenda');
  const [selectedDate, setSelectedDate] = useState(new Date());

  const agendaView = useMemo(() => {
    return posts
      .filter(post => isSameDay(parseISO(post.scheduledAt), selectedDate))
      .sort((a, b) => compareAsc(parseISO(a.scheduledAt), parseISO(b.scheduledAt)));
  }, [posts, selectedDate]);

  return (
    <div className="mobile-calendar">
      <div className="calendar-header">
        <button
          onClick={() => setView('agenda')}
          className={view === 'agenda' ? 'active' : ''}
        >
          Agenda
        </button>
        <button
          onClick={() => setView('month')}
          className={view === 'month' ? 'active' : ''}
        >
          Month
        </button>
      </div>

      {view === 'agenda' ? (
        <div className="agenda-view">
          <DatePicker
            selected={selectedDate}
            onChange={setSelectedDate}
            inline
            calendarClassName="mobile-datepicker"
          />
          <div className="posts-list">
            {agendaView.map(post => (
              <MobilePostCard
                key={post.id}
                post={post}
                onSelect={onPostSelect}
              />
            ))}
          </div>
        </div>
```

```
      ) : (
        <MonthView
          posts={posts}
          selectedDate={selectedDate}
          onDateSelect={setSelectedDate}
          onPostSelect={onPostSelect}
        />
      )}
    </div>
  );
};

// Touch-optimized post composer
const MobilePostComposer: React.FC = () => {
  const [activeTab, setActiveTab] = useState(0);
  const [content, setContent] = useState('');
  const [selectedPlatforms, setSelectedPlatforms] = useState<string[]>([]);

  const tabs = ['Content', 'Media', 'Schedule', 'Settings'];

  return (
    <div className="mobile-composer">
      <div className="composer-tabs">
        {tabs.map((tab, index) => (
          <button
            key={tab}
            onClick={() => setActiveTab(index)}
            className={`tab ${activeTab === index ? 'active' : ''}`}
          >
            {tab}
          </button>
        ))}
      </div>

      <div className="tab-content">
        {activeTab === 0 && (
          <ContentTab
            content={content}
            onChange={setContent}
            platforms={selectedPlatforms}
          />
        )}
        {activeTab === 1 && (
          <MediaTab />
        )}
        {activeTab === 2 && (
          <ScheduleTab />
```

```
        )}
        {activeTab === 3 && (
          <SettingsTab
            platforms={selectedPlatforms}
            onPlatformsChange={setSelectedPlatforms}
          />
        )}
      </div>

      <div className="mobile-actions">
        <button className="save-draft">Save Draft</button>
        <button className="schedule-post">Schedule</button>
      </div>
    </div>
  );
};
```

17. COMPLIANCE & SECURITY

17.1 GDPR Compliance

typescript

```
// Data privacy and GDPR compliance
class DataPrivacyService {
  async exportUserData(userId: string): Promise<UserDataExport> {
    // Collect all user data across services
    const userData = await this.collectUserData(userId);

    return {
      personal_information: userData.profile,
      posts: userData.posts,
      analytics: userData.analytics,
      workspace_memberships: userData.workspaces,
      export_date: new Date().toISOString(),
      retention_period: '7 years from last activity'
    };
  }

  async deleteUserData(userId: string, reason: string): Promise<void> {
    // Log deletion request
    await this.auditLogger.log({
      action: 'user_data_deletion',
      userId,
      reason,
      timestamp: new Date()
    });

    // Delete data across all services
    await Promise.all([
      this.deleteUserPosts(userId),
      this.deleteUserAnalytics(userId),
      this.deleteUserProfile(userId),
      this.deleteUserSessions(userId)
    ]);

    // Anonymize remaining references
    await this.anonymizeUserReferences(userId);
  }

  async handleConsentWithdrawal(userId: string, consentType: string): Promise<void> {
    switch (consentType) {
      case 'analytics':
        await this.stopAnalyticsCollection(userId);
        await this.deleteAnalyticsData(userId);
        break;
      case 'marketing':
        await this.removeFromMarketingLists(userId);
        break;
```

```
      case 'cookies':
        await this.deleteCookieData(userId);
        break;
    }
  }

  async anonymizeData(userId: string): Promise<void> {
    const anonymousId = `anon_${crypto.randomUUID()}`;

    // Replace PII with anonymous identifiers
    await db.users.update({
      where: { id: userId },
      data: {
        email: `${anonymousId}@anonymized.com`,
        first_name: 'Anonymous',
        last_name: 'User',
        avatar_url: null
      }
    });

    // Update posts to show anonymous creator
    await db.posts.updateMany({
      where: { created_by: userId },
      data: { created_by: anonymousId }
    });
  }
}

// Cookie consent management
class CookieConsentService {
  setConsentPreferences(userId: string, preferences: ConsentPreferences): void {
    const consent = {
      necessary: true, // Always required
      analytics: preferences.analytics || false,
      marketing: preferences.marketing || false,
      functional: preferences.functional || false,
      timestamp: new Date().toISOString()
    };

    // Store consent in database
    this.storeConsent(userId, consent);

    // Configure tracking based on consent
    this.configureTracking(consent);
  }

  private configureTracking(consent: ConsentPreferences): void {
```

```
    if (consent.analytics) {
      // Enable analytics tracking
      gtag('consent', 'update', {
        analytics_storage: 'granted'
      });
    } else {
      gtag('consent', 'update', {
        analytics_storage: 'denied'
      });
    }

    if (consent.marketing) {
      // Enable marketing tracking
      gtag('consent', 'update', {
        ad_storage: 'granted'
      });
    }
  }
}
```

17.2 SOC 2 Compliance

typescript

```
// Audit logging for SOC 2 compliance
class AuditLogger {
  async log(event: AuditEvent): Promise<void> {
    const auditRecord = {
      id: crypto.randomUUID(),
      timestamp: new Date(),
      event_type: event.type,
      user_id: event.userId,
      resource_id: event.resourceId,
      resource_type: event.resourceType,
      action: event.action,
      result: event.result,
      ip_address: event.ipAddress,
      user_agent: event.userAgent,
      details: event.details,
      risk_level: this.calculateRiskLevel(event)
    };

    // Store in immutable audit log
    await this.auditStore.insert(auditRecord);

    // Send high-risk events to SIEM
    if (auditRecord.risk_level === 'HIGH') {
      await this.siemService.sendAlert(auditRecord);
    }
  }

  private calculateRiskLevel(event: AuditEvent): 'LOW' | 'MEDIUM' | 'HIGH' {
    const highRiskActions = ['user_deletion', 'workspace_deletion', 'admin_access'];
    const mediumRiskActions = ['login_failure', 'password_change', 'permission_change'];

    if (highRiskActions.includes(event.action)) return 'HIGH';
    if (mediumRiskActions.includes(event.action)) return 'MEDIUM';
    return 'LOW';
  }

  async generateAuditReport(startDate: Date, endDate: Date): Promise<AuditReport> {
    const events = await this.auditStore.findByDateRange(startDate, endDate);

    return {
      period: { start: startDate, end: endDate },
      total_events: events.length,
      events_by_type: this.groupEventsByType(events),
      high_risk_events: events.filter(e => e.risk_level === 'HIGH'),
      failed_login_attempts: events.filter(e => e.action === 'login_failure').length,
      data_access_events: events.filter(e => e.action.includes('data_access')),
```

```
      generated_at: new Date()
    };
  }
}

// Access control and permissions
class AccessControlService {
  async checkPermission(
    userId: string,
    resource: string,
    action: string,
    context?: any
  ): Promise<boolean> {
    // Get user roles and permissions
    const userRoles = await this.getUserRoles(userId, context?.workspaceId);

    // Check direct permissions
    for (const role of userRoles) {
      const permissions = await this.getRolePermissions(role.id);

      if (this.hasPermission(permissions, resource, action)) {
        await this.auditLogger.log({
          type: 'access_granted',
          userId,
          action: `${action}_${resource}`,
          result: 'success',
          details: { roleId: role.id }
        });
        return true;
      }
    }

    // Check resource-specific permissions
    if (context?.resourceId) {
      const resourcePermissions = await this.getResourcePermissions(
        userId,
        resource,
        context.resourceId
      );

      if (this.hasPermission(resourcePermissions, resource, action)) {
        return true;
      }
    }

    await this.auditLogger.log({
      type: 'access_denied',
```

```
      userId,
      action: `${action}_${resource}`,
      result: 'failure',
      details: { reason: 'insufficient_permissions' }
    });

    return false;
  }

  private hasPermission(permissions: Permission[], resource: string, action: string): boolean {
    return permissions.some(p =>
      (p.resource === resource || p.resource === '*') &&
      (p.action === action || p.action === '*')
    );
  }
}
```

18. DEPLOYMENT CHECKLIST

18.1 Pre-Production Checklist

markdown

Security Checklist
- [ ] All API endpoints require authentication
- [ ] Rate limiting implemented on all public endpoints
- [ ] SQL injection protection in place
- [ ] XSS protection headers configured
- [ ] CSRF protection implemented
- [ ] Sensitive data encrypted at rest
- [ ] TLS/SSL certificates configured
- [ ] Security headers (HSTS, CSP, etc.) configured
- [ ] Dependency vulnerabilities scanned and resolved
- [ ] Secrets management system in place

Performance Checklist
- [ ] Database indexes optimized
- [ ] Caching strategy implemented
- [ ] CDN configured for static assets
- [ ] Image optimization and compression
- [ ] API response times under 200ms for 95th percentile
- [ ] Database query performance optimized
- [ ] Memory usage monitored and optimized
- [ ] Load testing completed successfully

Monitoring Checklist
- [ ] Application metrics collection (Prometheus/Grafana)
- [ ] Error tracking and alerting (Sentry)
- [ ] Log aggregation system configured
- [ ] Health check endpoints implemented
- [ ] Uptime monitoring configured
- [ ] Performance monitoring dashboards created
- [ ] Alert thresholds configured for critical metrics
- [ ] On-call rotation and escalation procedures defined

Data & Backup Checklist
- [ ] Database backup strategy implemented
- [ ] Backup restoration tested
- [ ] Data retention policies defined
- [ ] GDPR compliance measures implemented
- [ ] Data migration scripts tested
- [ ] Database connection pooling configured
- [ ] Read replicas configured for scaling

Infrastructure Checklist
- [ ] Auto-scaling policies configured
- [ ] Load balancer health checks configured
- [ ] Container orchestration (Kubernetes) configured
- [ ] Infrastructure as Code (Terraform) implemented

- [ ] Disaster recovery plan documented and tested
- [ ] Multi-region deployment strategy (if applicable)
- [ ] Network security groups configured
- [ ] VPC and subnet configuration secured

Social Media Integration Checklist
- [ ] All social media API credentials secured
- [ ] Rate limits for social media APIs respected
- [ ] Webhook endpoints secured and validated
- [ ] Token refresh mechanisms implemented
- [ ] Error handling for API failures
- [ ] Fallback strategies for API outages
- [ ] Platform-specific content validation

Testing Checklist
- [ ] Unit test coverage > 80%
- [ ] Integration tests for critical paths
- [ ] End-to-end tests for user journeys
- [ ] Load testing for expected traffic
- [ ] Security penetration testing completed
- [ ] Cross-browser compatibility testing
- [ ] Mobile responsiveness testing
- [ ] Accessibility compliance testing (WCAG 2.1)

18.2 Go-Live Procedures

typescript

```
// Deployment automation script
class DeploymentManager {
  private kubectl: KubectlClient;
  private terraform: TerraformClient;
  private healthChecker: HealthCheckService;

  async deployToProduction(version: string): Promise<DeploymentResult> {
    const deploymentId = crypto.randomUUID();

    try {
      console.log(`🚀 Starting deployment ${deploymentId} for version ${version}`);

      // 1. Pre-deployment checks
      await this.runPreDeploymentChecks();

      // 2. Database migrations
      await this.runDatabaseMigrations();

      // 3. Deploy backend services
      await this.deployBackendServices(version);

      // 4. Deploy frontend
      await this.deployFrontend(version);

      // 5. Run smoke tests
      await this.runSmokeTests();

      // 6. Update load balancer
      await this.updateLoadBalancer();

      // 7. Post-deployment verification
      await this.verifyDeployment();

      console.log(`✅ Deployment ${deploymentId} completed successfully`);

      return {
        success: true,
        deploymentId,
        version,
        timestamp: new Date()
      };

    } catch (error) {
      console.error(`❌ Deployment ${deploymentId} failed:`, error);

      // Automatic rollback
```

```
    await this.rollback();

    throw new Error(`Deployment failed: ${error.message}`);
  }
}

private async runPreDeploymentChecks(): Promise<void> {
  console.log('🔍 Running pre-deployment checks...');

  // Check system health
  const health = await this.healthChecker.getOverallHealth();
  if (health.status !== 'healthy') {
    throw new Error('System is not healthy for deployment');
  }

  // Check resource availability
  const resources = await this.kubectl.getClusterResources();
  if (resources.cpu.usage > 0.8 || resources.memory.usage > 0.8) {
    throw new Error('Insufficient cluster resources for deployment');
  }

  // Check dependencies
  await this.checkExternalDependencies();
}

private async runDatabaseMigrations(): Promise<void> {
  console.log('📊 Running database migrations...');

  // Create backup before migration
  await this.createDatabaseBackup();

  // Run migrations
  const migrationResult = await this.executeMigrations();

  if (!migrationResult.success) {
    throw new Error(`Migration failed: ${migrationResult.error}`);
  }

  console.log(`✅ Applied ${migrationResult.migrationsCount} migrations`);
}

private async deployBackendServices(version: string): Promise<void> {
  console.log('🔧 Deploying backend services...');

  const services = [
    'api-gateway',
    'auth-service',
```

```
    'content-service',
    'social-integration-service',
    'analytics-service',
    'notification-service'
  ];

  for (const service of services) {
    await this.deployService(service, version);
    await this.waitForServiceReady(service);
  }
}

private async deployService(serviceName: string, version: string): Promise<void> {
  const manifest = {
    apiVersion: 'apps/v1',
    kind: 'Deployment',
    metadata: {
      name: serviceName,
      labels: { app: serviceName, version }
    },
    spec: {
      replicas: 3,
      selector: { matchLabels: { app: serviceName } },
      template: {
        metadata: { labels: { app: serviceName, version } },
        spec: {
          containers: [{
            name: serviceName,
            image: `your-registry/${serviceName}:${version}`,
            ports: [{ containerPort: 3000 }],
            env: await this.getServiceEnvironment(serviceName),
            resources: {
              requests: { memory: '256Mi', cpu: '250m' },
              limits: { memory: '512Mi', cpu: '500m' }
            },
            livenessProbe: {
              httpGet: { path: '/health/live', port: 3000 },
              initialDelaySeconds: 30,
              periodSeconds: 10
            },
            readinessProbe: {
              httpGet: { path: '/health/ready', port: 3000 },
              initialDelaySeconds: 5,
              periodSeconds: 5
            }
          }]
        }
```

```
      }
    }
  };

  await this.kubectl.apply(manifest);
 }

 private async runSmokeTests(): Promise<void> {
  console.log('🧪 Running smoke tests...');

  const tests = [
   () => this.testHealthEndpoints(),
   () => this.testAuthentication(),
   () => this.testCriticalPaths(),
   () => this.testSocialMediaIntegration()
  ];

  for (const test of tests) {
   const result = await test();
   if (!result.success) {
    throw new Error(`Smoke test failed: ${result.error}`);
   }
  }
 }

 private async rollback(): Promise<void> {
  console.log('🔄 Initiating rollback...');

  // Get previous stable version
  const previousVersion = await this.getPreviousStableVersion();

  // Rollback services
  await this.deployToProduction(previousVersion);

  // Restore database if needed
  await this.restoreDatabaseIfNeeded();

  console.log('✅ Rollback completed');
 }
}

// Blue-green deployment strategy
class BlueGreenDeployment {
 async deploy(newVersion: string): Promise<void> {
  const currentEnvironment = await this.getCurrentEnvironment(); // 'blue' or 'green'
  const targetEnvironment = currentEnvironment === 'blue' ? 'green' : 'blue';
```

```
    console.log(`Deploying ${newVersion} to ${targetEnvironment} environment`);

    // 1. Deploy to target environment
    await this.deployToEnvironment(targetEnvironment, newVersion);

    // 2. Run health checks on target environment
    await this.healthCheckEnvironment(targetEnvironment);

    // 3. Run smoke tests on target environment
    await this.smokeTestEnvironment(targetEnvironment);

    // 4. Switch traffic to target environment
    await this.switchTraffic(targetEnvironment);

    // 5. Monitor for issues
    await this.monitorDeployment(targetEnvironment);

    // 6. Keep old environment for quick rollback (optional)
    console.log(`Deployment complete. ${targetEnvironment} is now active.`);
  }

  private async switchTraffic(targetEnvironment: string): Promise<void> {
    // Update load balancer configuration
    await this.updateLoadBalancerConfig({
      target: targetEnvironment,
      healthCheck: `/health`,
      rollbackThreshold: 0.95 // Rollback if success rate drops below 95%
    });

    // Gradual traffic shift (canary-style)
    const trafficPercentages = [10, 25, 50, 75, 100];

    for (const percentage of trafficPercentages) {
      await this.setTrafficPercentage(targetEnvironment, percentage);
      await this.sleep(60000); // Wait 1 minute

      const healthMetrics = await this.getEnvironmentMetrics(targetEnvironment);
      if (healthMetrics.errorRate > 0.05) { // More than 5% error rate
        throw new Error('High error rate detected, aborting traffic switch');
      }
    }
  }
}
```

19. MAINTENANCE & SUPPORT

19.1 Automated Maintenance Tasks

typescript

```
// Automated maintenance scheduler
class MaintenanceScheduler {
  private cron = require('node-cron');

  constructor() {
    this.scheduleTasks();
  }

  private scheduleTasks(): void {
    // Daily tasks
    this.cron.schedule('0 2 * * *', async () => {
      await this.runDailyMaintenance();
    });

    // Weekly tasks
    this.cron.schedule('0 3 * * 0', async () => {
      await this.runWeeklyMaintenance();
    });

    // Monthly tasks
    this.cron.schedule('0 4 1 * *', async () => {
      await this.runMonthlyMaintenance();
    });
  }

  private async runDailyMaintenance(): Promise<void> {
    console.log('🔧 Running daily maintenance tasks...');

    await Promise.all([
      this.cleanupExpiredSessions(),
      this.archiveOldAnalytics(),
      this.cleanupTempFiles(),
      this.updateSocialMediaMetrics(),
      this.validateSystemHealth()
    ]);

    console.log('✅ Daily maintenance completed');
  }

  private async runWeeklyMaintenance(): Promise<void> {
    console.log('🔧 Running weekly maintenance tasks...');

    await Promise.all([
      this.optimizeDatabaseIndexes(),
      this.cleanupOldLogs(),
      this.updateSecurityPatches(),
```

```
    this.generateWeeklyReports(),
    this.backupCriticalData()
  ]);

  console.log('✅ Weekly maintenance completed');
}

private async runMonthlyMaintenance(): Promise<void> {
  console.log('🔧 Running monthly maintenance tasks...');

  await Promise.all([
    this.archiveOldData(),
    this.renewSSLCertificates(),
    this.performSecurityAudit(),
    this.optimizeStorageUsage(),
    this.updateDependencies()
  ]);

  console.log('✅ Monthly maintenance completed');
}

private async cleanupExpiredSessions(): Promise<void> {
  const expiredCount = await db.user_sessions.deleteMany({
    where: {
      expires_at: { lt: new Date() }
    }
  });

  console.log(`Cleaned up ${expiredCount.count} expired sessions`);
}

private async archiveOldAnalytics(): Promise<void> {
  const cutoffDate = new Date();
  cutoffDate.setMonth(cutoffDate.getMonth() - 6); // Archive data older than 6 months

  const oldAnalytics = await db.post_analytics.findMany({
    where: {
      collected_at: { lt: cutoffDate }
    }
  });

  if (oldAnalytics.length > 0) {
    // Move to archive storage (S3 Glacier)
    await this.archiveToS3(oldAnalytics, 'analytics-archive');

    // Delete from primary database
    await db.post_analytics.deleteMany({
```

```
      where: {
        collected_at: { lt: cutoffDate }
      }
    });

    console.log(`Archived ${oldAnalytics.length} old analytics records`);
  }
}

private async optimizeDatabaseIndexes(): Promise<void> {
  const queries = [
    'REINDEX INDEX CONCURRENTLY idx_posts_workspace_id;',
    'REINDEX INDEX CONCURRENTLY idx_posts_scheduled_at;',
    'ANALYZE posts;',
    'ANALYZE post_analytics;',
    'ANALYZE workspaces;'
  ];

  for (const query of queries) {
    await db.$executeRawUnsafe(query);
  }

  console.log('Database indexes optimized');
}
}

// System health monitoring and alerting
class SystemHealthMonitor {
private alertingService: AlertingService;
private metricsCollector: MetricsCollector;

constructor() {
  this.alertingService = new AlertingService();
  this.metricsCollector = new MetricsCollector();
  this.startMonitoring();
}

private startMonitoring(): void {
  // Check system health every minute
  setInterval(async () => {
    await this.checkSystemHealth();
  }, 60000);

  // Check critical metrics every 5 minutes
  setInterval(async () => {
    await this.checkCriticalMetrics();
  }, 300000);
```

```
}

private async checkSystemHealth(): Promise<void> {
  const healthChecks = await Promise.allSettled([
    this.checkDatabaseHealth(),
    this.checkRedisHealth(),
    this.checkAPIHealth(),
    this.checkSocialMediaAPIs()
  ]);

  const failures = healthChecks.filter(result => result.status === 'rejected');

  if (failures.length > 0) {
    await this.alertingService.sendAlert({
      severity: 'high',
      title: 'System Health Check Failed',
      message: `${failures.length} health checks failed`,
      details: failures.map(f => f.reason)
    });
  }
}

private async checkCriticalMetrics(): Promise<void> {
  const metrics = await this.metricsCollector.getCurrentMetrics();

  // Check error rates
  if (metrics.errorRate > 0.05) { // 5% error rate threshold
    await this.alertingService.sendAlert({
      severity: 'high',
      title: 'High Error Rate Detected',
      message: `Current error rate: ${(metrics.errorRate * 100).toFixed(2)}%`
    });
  }

  // Check response times
  if (metrics.averageResponseTime > 2000) { // 2 second threshold
    await this.alertingService.sendAlert({
      severity: 'medium',
      title: 'High Response Time',
      message: `Average response time: ${metrics.averageResponseTime}ms`
    });
  }

  // Check memory usage
  if (metrics.memoryUsage > 0.9) { // 90% memory usage
    await this.alertingService.sendAlert({
      severity: 'high',
```

```
        title: 'High Memory Usage',
        message: `Memory usage: ${(metrics.memoryUsage * 100).toFixed(2)}%`
      });
    }

    // Check disk space
    if (metrics.diskUsage > 0.85) { // 85% disk usage
      await this.alertingService.sendAlert({
        severity: 'medium',
        title: 'High Disk Usage',
        message: `Disk usage: ${(metrics.diskUsage * 100).toFixed(2)}%`
      });
    }
  }
}
```

19.2 Support System Integration

typescript

```
// Customer support integration
class SupportSystem {
  private ticketingSystem: TicketingService;
  private knowledgeBase: KnowledgeBaseService;
  private liveChatService: LiveChatService;

  async createSupportTicket(request: SupportRequest): Promise<SupportTicket> {
    const ticket = await this.ticketingSystem.create({
      title: request.title,
      description: request.description,
      priority: this.calculatePriority(request),
      category: request.category,
      userId: request.userId,
      workspaceId: request.workspaceId,
      attachments: request.attachments
    });

    // Auto-assign based on category and severity
    const assignee = await this.findBestAssignee(ticket);
    if (assignee) {
      await this.ticketingSystem.assign(ticket.id, assignee.id);
    }

    // Send confirmation email
    await this.emailService.sendTicketConfirmation(ticket);

    return ticket;
  }

  private calculatePriority(request: SupportRequest): TicketPriority {
    // Auto-categorize based on keywords and user tier
    const urgentKeywords = ['down', 'not working', 'emergency', 'critical'];
    const hasUrgentKeywords = urgentKeywords.some(keyword =>
      request.description.toLowerCase().includes(keyword)
    );

    if (hasUrgentKeywords && request.userTier === 'enterprise') {
      return TicketPriority.CRITICAL;
    }

    if (hasUrgentKeywords || request.userTier === 'enterprise') {
      return TicketPriority.HIGH;
    }

    return TicketPriority.MEDIUM;
  }
```

```
  async provideSelfServiceSolution(query: string): Promise<SelfServiceResult> {
    // Search knowledge base
    const articles = await this.knowledgeBase.search(query);

    // Find similar resolved tickets
    const similarTickets = await this.ticketingSystem.findSimilar(query);

    // Generate AI-powered suggestions
    const aiSuggestions = await this.generateAISuggestions(query);

    return {
      articles: articles.slice(0, 5),
      similarTickets: similarTickets.slice(0, 3),
      aiSuggestions,
      escalationAvailable: true
    };
  }

  private async generateAISuggestions(query: string): Promise<string[]> {
    // Use AI/ML to generate contextual suggestions
    // This could integrate with OpenAI or similar service
    const suggestions = [
      "Try refreshing your browser and clearing cache",
      "Check if you have the required permissions for this action",
      "Verify your social media account connections in Settings"
    ];

    return suggestions;
  }
}

// In-app help and onboarding
class OnboardingService {
  async getOnboardingFlow(userId: string): Promise<OnboardingStep[]> {
    const user = await this.getUserProfile(userId);
    const completedSteps = await this.getCompletedSteps(userId);

    const allSteps: OnboardingStep[] = [
      {
        id: 'connect-social-accounts',
        title: 'Connect Your Social Media Accounts',
        description: 'Link your Facebook, Twitter, LinkedIn and other accounts',
        component: 'SocialAccountSetup',
        required: true
      },
      {
```

```
      id: 'create-first-post',
      title: 'Create Your First Post',
      description: 'Learn how to compose and schedule content',
      component: 'PostComposerTour',
      required: true
    },
    {
      id: 'setup-approval-workflow',
      title: 'Set Up Approval Workflow',
      description: 'Configure content approval process for your team',
      component: 'ApprovalWorkflowSetup',
      required: false,
      condition: () => user.role === 'workspace_admin'
    },
    {
      id: 'invite-team-members',
      title: 'Invite Team Members',
      description: 'Add colleagues to collaborate on content',
      component: 'TeamInvitation',
      required: false
    },
    {
      id: 'explore-analytics',
      title: 'Explore Analytics',
      description: 'Learn how to track your content performance',
      component: 'AnalyticsTour',
      required: false
    }
  ];

  return allSteps.filter(step =>
    !completedSteps.includes(step.id) &&
    (!step.condition || step.condition())
  );
}

async markStepCompleted(userId: string, stepId: string): Promise<void> {
  await db.user_onboarding.upsert({
    where: { userId_stepId: { userId, stepId } },
    update: { completedAt: new Date() },
    create: { userId, stepId, completedAt: new Date() }
  });

  // Check if onboarding is complete
  const remainingSteps = await this.getOnboardingFlow(userId);
  const requiredSteps = remainingSteps.filter(step => step.required);
```

```
    if (requiredSteps.length === 0) {
      await this.markOnboardingComplete(userId);
    }
  }

  async provideContextualHelp(page: string, userAction: string): Promise<HelpContent> {
    const helpMap: Record<string, HelpContent> = {
      'post-composer': {
        title: 'Creating Posts',
        content: 'Learn how to create engaging content for multiple platforms',
        videoUrl: '/help/videos/post-composer.mp4',
        articles: ['creating-posts', 'platform-optimization', 'content-guidelines']
      },
      'calendar': {
        title: 'Content Calendar',
        content: 'Organize and schedule your content effectively',
        videoUrl: '/help/videos/calendar.mp4',
        articles: ['scheduling-posts', 'calendar-views', 'bulk-operations']
      },
      'analytics': {
        title: 'Analytics Dashboard',
        content: 'Track and analyze your social media performance',
        videoUrl: '/help/videos/analytics.mp4',
        articles: ['understanding-metrics', 'creating-reports', 'performance-optimization']
      }
    };

    return helpMap[page] || {
      title: 'Help',
      content: 'Need assistance? Check our knowledge base or contact support.',
      articles: ['getting-started', 'common-issues', 'contact-support']
    };
  }
}
```

20. FINAL IMPLEMENTATION SUMMARY

20.1 Development Timeline Summary

markdown

Implementation Timeline (24 Weeks Total)

Phase 1: Foundation (Weeks 1-4)
- ✅ Infrastructure setup and CI/CD pipeline
- ✅ Database design and authentication system
- ✅ Basic API structure and security measures
- ✅ Development environment configuration

Phase 2: Core Features (Weeks 5-12)
- ✅ Content creation and management system
- ✅ Social media platform integrations
- ✅ Scheduling and publishing engine
- ✅ Multi-platform content optimization

Phase 3: Collaboration (Weeks 13-16)
- ✅ Approval workflow system
- ✅ Real-time collaboration features
- ✅ Comment and feedback system
- ✅ Role-based access control

Phase 4: Analytics & Reporting (Weeks 17-20)
- ✅ Data collection and processing
- ✅ Analytics dashboard and insights
- ✅ Custom report generation
- ✅ Performance optimization recommendations

Phase 5: Enterprise Features (Weeks 21-24)
- ✅ White-label capabilities
- ✅ Advanced security and compliance
- ✅ Scalability improvements
- ✅ Mobile optimization and PWA

20.2 Key Technologies and Architecture Decisions

typescript

```
// Final architecture overview
const TechnologyStack = {
  Frontend: {
    framework: 'React 18 with TypeScript',
    stateManagement: 'Zustand',
    styling: 'Tailwind CSS',
    bundler: 'Vite',
    pwa: 'Workbox',
    testing: 'Jest + React Testing Library + Playwright'
  },

  Backend: {
    runtime: 'Node.js 18+',
    framework: 'Express.js with TypeScript',
    database: 'PostgreSQL 15',
    caching: 'Redis 7',
    queue: 'Bull Queue with Redis',
    fileStorage: 'AWS S3',
    monitoring: 'Prometheus + Grafana'
  },

  Infrastructure: {
    containerization: 'Docker + Kubernetes',
    cloudProvider: 'AWS',
    iac: 'Terraform',
    cicd: 'GitHub Actions',
    monitoring: 'DataDog / New Relic',
    logging: 'Winston + ELK Stack'
  },

  Integrations: {
    socialPlatforms: [
      'Facebook Graph API',
      'Twitter API v2',
      'LinkedIn API',
      'Instagram Basic Display API',
      'TikTok Business API',
      'YouTube Data API',
      'Pinterest API'
    ],
    thirdPartyServices: [
      'SendGrid (Email)',
      'Twilio (SMS)',
      'Stripe (Payments)',
      'Auth0 (Identity)',
      'Sentry (Error Tracking)'
```

```
    ]
  }
};

// Scalability features implemented
const ScalabilityFeatures = {
  database: {
    sharding: 'Horizontal sharding by workspace',
    readReplicas: 'Multiple read replicas for analytics',
    caching: 'Multi-layer caching (Redis + in-memory)',
    partitioning: 'Time-based partitioning for analytics data'
  },

  application: {
    microservices: 'Service-oriented architecture',
    eventDriven: 'Event sourcing with Redis Streams',
    loadBalancing: 'Application load balancer with health checks',
    autoScaling: 'Horizontal pod autoscaling based on CPU/memory'
  },

  performance: {
    cdn: 'CloudFront for static assets',
    imageOptimization: 'WebP format with fallbacks',
    codesplitting: 'Route-based code splitting',
    lazyLoading: 'Virtual scrolling for large lists'
  }
};

// Security measures implemented
const SecurityFeatures = {
  authentication: {
    method: 'JWT with refresh token rotation',
    mfa: '2FA support with TOTP/SMS',
    oauth: 'Social login integration',
    passwordPolicy: 'Strong password requirements'
  },

  authorization: {
    rbac: 'Role-based access control',
    permissions: 'Granular permission system',
    api: 'API key management for integrations'
  },

  dataProtection: {
    encryption: 'AES-256 encryption at rest',
    transmission: 'TLS 1.3 for data in transit',
    secrets: 'HashiCorp Vault for secret management',
```

```
    backup: 'Encrypted automated backups'
  },

  compliance: {
    gdpr: 'Full GDPR compliance with data portability',
    soc2: 'SOC 2 Type II compliance ready',
    audit: 'Comprehensive audit logging',
    privacy: 'Privacy-by-design implementation'
  }
};
```

20.3 Launch Preparation

typescript

```
// Production deployment configuration
const ProductionConfig = {
  environment: {
    DATABASE_URL: 'postgresql://user:pass@prod-db:5432/planable_prod',
    REDIS_URL: 'redis://prod-redis:6379',
    AWS_REGION: 'us-east-1',
    CDN_URL: 'https://cdn.planable-clone.com',
    API_URL: 'https://api.planable-clone.com'
  },

  scaling: {
    minReplicas: 3,
    maxReplicas: 50,
    targetCPU: 70,
    targetMemory: 80
  },

  monitoring: {
    uptime: 'UptimeRobot',
    apm: 'DataDog',
    logs: 'CloudWatch Logs',
    alerts: 'PagerDuty integration'
  },

  backups: {
    database: 'Daily automated backups with 30-day retention',
    files: 'Versioned S3 storage with lifecycle policies',
    disaster: 'Multi-region backup strategy'
  }
};

// Go-live checklist verification
const GoLiveChecklist = {
  technical: [
    '✅ All tests passing (unit, integration, e2e)',
    '✅ Performance benchmarks met',
    '✅ Security audit completed',
    '✅ Load testing successful',
    '✅ Monitoring and alerting configured',
    '✅ Backup and recovery tested',
    '✅ SSL certificates valid',
    '✅ CDN configuration verified'
  ],

  business: [
    '✅ Legal terms and privacy policy ready',
```

```
    '✅ Pricing and billing system tested',
    '✅ Customer support processes defined',
    '✅ Documentation and help center complete',
    '✅ Marketing materials prepared',
    '✅ User onboarding flow tested',
    '✅ Beta user feedback incorporated',
    '✅ Launch strategy finalized'
  ],

  operational: [
    '✅ On-call schedule established',
    '✅ Incident response procedures documented',
    '✅ Escalation paths defined',
    '✅ Team training completed',
    '✅ Communication channels set up',
    '✅ Rollback procedures tested',
    '✅ Capacity planning completed',
    '✅ Cost monitoring configured'
  ]
};
```

This comprehensive development plan provides a complete roadmap for building a Planable-like social media management platform with all pro features. The plan includes detailed technical specifications, implementation strategies, security measures, scalability considerations, and operational procedures necessary for a production-ready enterprise application.

The modular approach allows for iterative development while maintaining code quality and system reliability. Each phase builds upon the previous one, ensuring a stable foundation for advanced features and enterprise-grade scalability.