To build the **LUMINA Care OS** SaaS platform with **React (Vite, JavaScript)** for the frontend and **Python Django** for the backend, I'll outline the technologies, frameworks, and tools required across the frontend, backend, and supporting infrastructure. I'll also provide a developer-ready specification with code artifacts for key components, including a sample React frontend setup, Django backend setup, and an example API contract. The response will focus on delivering a clear, actionable plan for developers, addressing the core pillars and technical requirements outlined in the input.

Technology Stack Overview

Frontend (React with Vite, JavaScript)

- **Framework**: React (v18.x) with Vite for fast development and build.
- **Styling**: Tailwind CSS for responsive, utility-first design.
- **State Management**: React Query for data fetching, Zustand for lightweight state management.
- **UI Components**: Shadcn/UI or Ant Design for pre-built, customizable components.
- **Routing**: React Router for client-side navigation.
- **Mobile Strategy**: React Native for mobile apps, Progressive Web App (PWA) for fallback.
- **Internationalization**: i18next for multi-language support.
- **Forms**: React Hook Form for dynamic, performant forms.
- **Authentication**: JWT-based auth with refresh tokens, integrated with OAuth2 (Okta, Azure AD).
- **Real-time**: WebSocket (via Django Channels on backend) for notifications and live updates.
- **Accessibility**: WCAG 2.1 compliance with ARIA support via React Aria.

Backend (Python Django)

- **Framework**: Django (v5.x) with Django REST Framework (DRF) for RESTful APIs.
- **Database**: PostgreSQL for relational data, with schema-level multi-tenancy.
- **ORM**: Django ORM for data modeling, migrations, and tenant isolation.
- **Task Queue**: Celery with Redis for async tasks (e.g., notifications, AI processing).
- **Real-time**: Django Channels for WebSocket-based notifications.

- **Authentication**: Django Allauth for OAuth2/SSO, JWT for API auth.
- **File Storage**: AWS S3 or Azure Blob Storage for documents and assets.
- **API Documentation**: DRF-Spectacular for OpenAPI/Swagger generation.
- **Caching**: Redis for caching API responses and session data.
- **Search**: Elasticsearch for full-text search (e.g., CV parsing, document search).
- **AI Integration**: Python-based ML libraries (e.g., scikit-learn, spaCy) for CV parsing, sentiment analysis, and care plan suggestions.

Infrastructure & DevOps

- **Cloud**: AWS or Azure for cloud-native deployment.
- **Containerization**: Docker for services, Kubernetes for orchestration and auto-scaling.
- **CI/CD**: GitHub Actions or GitLab CI for automated testing, deployment, and rollback.
- **Monitoring**: Prometheus + Grafana for metrics, Sentry for error tracking.
- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging.
- **Security**: AWS WAF, Cloudflare for DDoS protection, and SSL/TLS via Let's Encrypt.
- **Backup**: Daily snapshots with AWS RDS or Azure Backup, cold storage for long-term retention.

AI & Automation

- **ML Frameworks**: TensorFlow or PyTorch for custom models (e.g., candidate matching, predictive care gaps).
- **NLP**: spaCy or Hugging Face Transformers for CV parsing, sentiment analysis, and note-to-text conversion.
- **Workflow Automation**: Temporal or Apache Airflow for orchestrating complex workflows (e.g., onboarding, compliance checks).
- **Notification System**: Firebase or OneSignal for push notifications, Amazon SES for email, Twilio for SMS.

Integrations

- **Third-Party APIs**: DBS, DVLA, Xero, Zoom, Teams, job boards (via REST APIs).
- **SSO**: Okta, Azure AD, Google Workspace via OAuth2/SAML2.
- **HRIS/ERP**: Workday, BambooHR, Sage via custom API connectors.

Compliance & Standards

- **Security**: ISO 27001, GDPR, with RBAC, MFA, and encrypted storage.
- **Audit**: ISO 9001 evidence generation via audit logs and dashboards.
- **Healthcare**: CQC-compliant care plans, eMAR, and risk flagging.

Developer-Ready Artifacts

Below are the key artifacts to kickstart development, including a React frontend setup, a Django backend setup, and an example API contract. These artifacts address the core requirements for the **Recruitment CRM** and **Care Coordination** modules, with a focus on multi-tenancy, RBAC, and API-driven workflows.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>LUMINA Care OS</title>
  <script src="https://cdn.jsdelivr.net/npm/react@18/umd/react.development.js"
  <script src="https://cdn.jsdelivr.net/npm/react-dom@18/umd/react-dom.develop
  <script src="https://cdn.tailwindcss.com"></script>
</head>
<body>
  <div id="root"></div>
  <script type="module">
    import React, { useState, useEffect } from 'https://cdn.jsdelivr.net/npm/r
    import ReactDOM from 'https://cdn.jsdelivr.net/npm/react-dom@18/+esm';
    import { BrowserRouter, Routes, Route } from 'https://cdn.jsdelivr.net/npm
    import { QueryClient, QueryClientProvider } from 'https://cdn.jsdelivr.net

    const queryClient = new QueryClient();

    const App = () => {
      const [user, setUser] = useState(null);

      useEffect(() => {
        // Simulate fetching user data with RBAC
        fetch('/api/auth/user', { headers: { Authorization: `Bearer ${localSto
```

```
      .then(res => res.json())
      .then(data => setUser(data));
  }, []);

  return (
    <div className="min-h-screen bg-gray-100">
      <header className="bg-blue-600 text-white p-4">
        <h1 className="text-2xl">LUMINA Care OS</h1>
      </header>
      <main className="p-4">
        <Routes>
          <Route path="/" element={<Dashboard user={user} />} />
          <Route path="/recruitment" element={<RecruitmentCRM user={user}
          <Route path="/care-coordination" element={<CareCoordination user
        </Routes>
      </main>
    </div>
  );
};

const Dashboard = ({ user }) => (
  <div>
    <h2 className="text-xl">Welcome, {user?.name || 'Guest'}</h2>
    <p>Role: {user?.role || 'N/A'}</p>
  </div>
);

const RecruitmentCRM = ({ user }) => (
  <div>
    <h2 className="text-xl">Recruitment CRM</h2>
    {user?.role === 'admin' && (
      <button className="bg-blue-500 text-white px-4 py-2 rounded">
        Create Job Posting
      </button>
    )}
  </div>
);

const CareCoordination = ({ user }) => (
  <div>
    <h2 className="text-xl">Care Coordination</h2>
    {user?.role === 'carer' && (
      <div>
```

```
              <p>View your assigned shifts and care plans.</p>
              <button className="bg-green-500 text-white px-4 py-2 rounded">
                Log Care Note
              </button>
            </div>
          )}
        </div>
      );

      ReactDOM.render(
        <QueryClientProvider client={queryClient}>
          <BrowserRouter>
            <App />
          </BrowserRouter>
        </QueryClientProvider>,
        document.getElementById('root')
      );
    </script>
</body>
</html>


#!/usr/bin/env python
import os
import sys

if __name__ == '__main__':
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'lumina.settings')
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)


import os
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent
```

```
SECRET_KEY = 'your-secret-key'
DEBUG = True
ALLOWED_HOSTS = ['*']

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'django_tenants',
    'channels',
    'core',
]

MIDDLEWARE = [
    'django_tenants.middleware.main.TenantMainMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'lumina.urls'
PUBLIC_SCHEMA_NAME = 'public'

TENANT_MODEL = 'core.Client'
TENANT_DOMAIN_MODEL = 'core.Domain'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
```

```
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

ASGI_APPLICATION = 'lumina.asgi.application'
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('redis', 6379)],
        },
    },
}

DATABASES = {
    'default': {
        'ENGINE': 'django_tenants.postgresql_backend',
        'NAME': 'lumina_db',
        'USER': 'postgres',
        'PASSWORD': 'your-password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}

STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

```
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'


from django.db import models
from django_tenants.models import TenantMixin, DomainMixin
from django.contrib.auth.models import AbstractUser

class Client(TenantMixin):
    name = models.CharField(max_length=100)
    created_on = models.DateField(auto_now_add=True)
    auto_create_schema = True

class Domain(DomainMixin):
    pass

class User(AbstractUser):
    role = models.CharField(max_length=50, choices=[
        ('admin', 'Admin'),
        ('carer', 'Carer'),
        ('client', 'Client'),
        ('family', 'Family'),
        ('auditor', 'Auditor'),
    ])
    tenant = models.ForeignKey(Client, on_delete=models.CASCADE, null=True)

class JobPosting(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    tenant = models.ForeignKey(Client, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    status = models.CharField(max_length=50, default='open')

class CarePlan(models.Model):
    client = models.ForeignKey('Client', on_delete=models.CASCADE)
    carer = models.ForeignKey(User, on_delete=models.CASCADE, limit_choices_to
    description = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    status = models.CharField(max_length=50, default='draft')


from rest_framework import viewsets, permissions
from rest_framework.decorators import action
from rest_framework.response import Response
```

```
from .models import JobPosting, CarePlan
from .serializers import JobPostingSerializer, CarePlanSerializer

class JobPostingViewSet(viewsets.ModelViewSet):
    queryset = JobPosting.objects.all()
    serializer_class = JobPostingSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        return self.queryset.filter(tenant=self.request.user.tenant)

    @action(detail=True, methods=['post'])
    def publish(self, request, pk=None):
        job = self.get_object()
        job.status = 'published'
        job.save()
        return Response({'status': 'Job published'})

class CarePlanViewSet(viewsets.ModelViewSet):
    queryset = CarePlan.objects.all()
    serializer_class = CarePlanSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        return self.queryset.filter(tenant=self.request.user.tenant)

from rest_framework import serializers
from .models import JobPosting, CarePlan

class JobPostingSerializer(serializers.ModelSerializer):
    class Meta:
        model = JobPosting
        fields = ['id', 'title', 'description', 'status', 'created_at']

class CarePlanSerializer(serializers.ModelSerializer):
    class Meta:
        model = CarePlan
        fields = ['id', 'client', 'carer', 'description', 'status', 'created_a


openapi: 3.0.3
info:
  title: LUMINA Care OS API
```

```yaml
  version: 1.0.0
paths:
  /api/job-postings/:
    get:
      summary: List job postings
      responses:
        '200':
          description: A list of job postings
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/JobPosting'
    post:
      summary: Create a job posting
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/JobPosting'
      responses:
        '201':
          description: Job posting created
  /api/care-plans/:
    get:
      summary: List care plans
      responses:
        '200':
          description: A list of care plans
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/CarePlan'
    post:
      summary: Create a care plan
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CarePlan'
```

```
      responses:
        '201':
          description: Care plan created
components:
  schemas:
    JobPosting:
      type: object
      properties:
        id:
          type: integer
        title:
          type: string
        description:
          type: string
        status:
          type: string
        created_at:
          type: string
          format: date-time
    CarePlan:
      type: object
      properties:
        id:
          type: integer
        client:
          type: integer
        carer:
          type: integer
        description:
          type: string
        status:
          type: string
        created_at:
          type: string
          format: date-time
```

Additional Developer Guidance

Data Architecture

- **Entities**:
  - **Client**: Tenant-specific organization (name, created_on).
  - **User**: Extends Django's AbstractUser with role and tenant fields.
  - **JobPosting**: Job details with tenant and status.
  - **CarePlan**: Client-carer linked care plans with status.
- **Multi-Tenancy**: Schema-level using `django-tenants` (each tenant has its own database schema).
- **Relationships**:
  - User ↔ Client (Many-to-One, tenant-based).
  - JobPosting ↔ Client (Many-to-One).
  - CarePlan ↔ Client, Carer (Many-to-One).

Key Workflows

1. **Recruitment Journey**:
   - Job post creation (Admin) → API push to job boards → Candidate applies → AI CV parsing → Interview scheduling → Offer generation → Onboarding tasks.
2. **Care Coordination**:
   - Client assessment → AI care plan generation → Carer assignment → Shift scheduling → Care note logging → Incident flagging.
3. **Compliance Check**:
   - Document upload → API-based DBS verification → Expiry tracking → Audit log generation.

Permissions Matrix

Role	Job Posting (View/Create/Edit/Delete)	Care Plan (View/Create/Edit/Delete)	Compliance (View/Edit)
Admin	All	All	All
Carer	View (own)	View/Edit (assigned)	View (own)
Client	None	View (own)	None
Family	None	View (linked client)	None
Auditor	View	View	View

AI Logic

- **CV Parsing**: Use spaCy to extract skills, experience, and gaps; score candidates based on job requirements.
- **Care Plan Suggestions**: Use pre-trained models to suggest care tasks based on client assessments.
- **Notifications**: Triggered via Celery tasks (e.g., “DBS expiring” email via Amazon SES).

Next Steps

1. Set up Vite project with `npm create vite@latest` and install dependencies.
2. Initialize Django project with `django-admin startproject lumina` and configure `django-tenants`.
3. Define CI/CD pipeline with GitHub Actions for automated testing and deployment.
4. Implement AI pipelines using Python scripts for CV parsing and care plan generation.
5. Create Swagger UI for API documentation using DRF-Spectacular.

This setup provides a solid foundation for building the **LUMINA Care OS**, with modular, scalable components and clear developer instructions. Let me know if you need further details or additional artifacts!