

cUPido - Documentación de Proyecto


Tags: #PROYECTO #DATING-APP #DJANGO #REACT #ARQUITECTURA #UNIVERSIDAD

Fecha: 2025-10-05

Estado: en-desarrollo



Proyecto cUPido - Aplicación de Citas Universitaria

 **Resumen Ejecutivo** Aplicación web de citas estilo Tinder exclusiva para la comunidad universitaria de la Universidad de Pamplona. Verificación por correo institucional, matching basado en likes mutuos, chat en tiempo real y arquitectura moderna full-stack.

Objetivos del Proyecto

Objetivos Principales

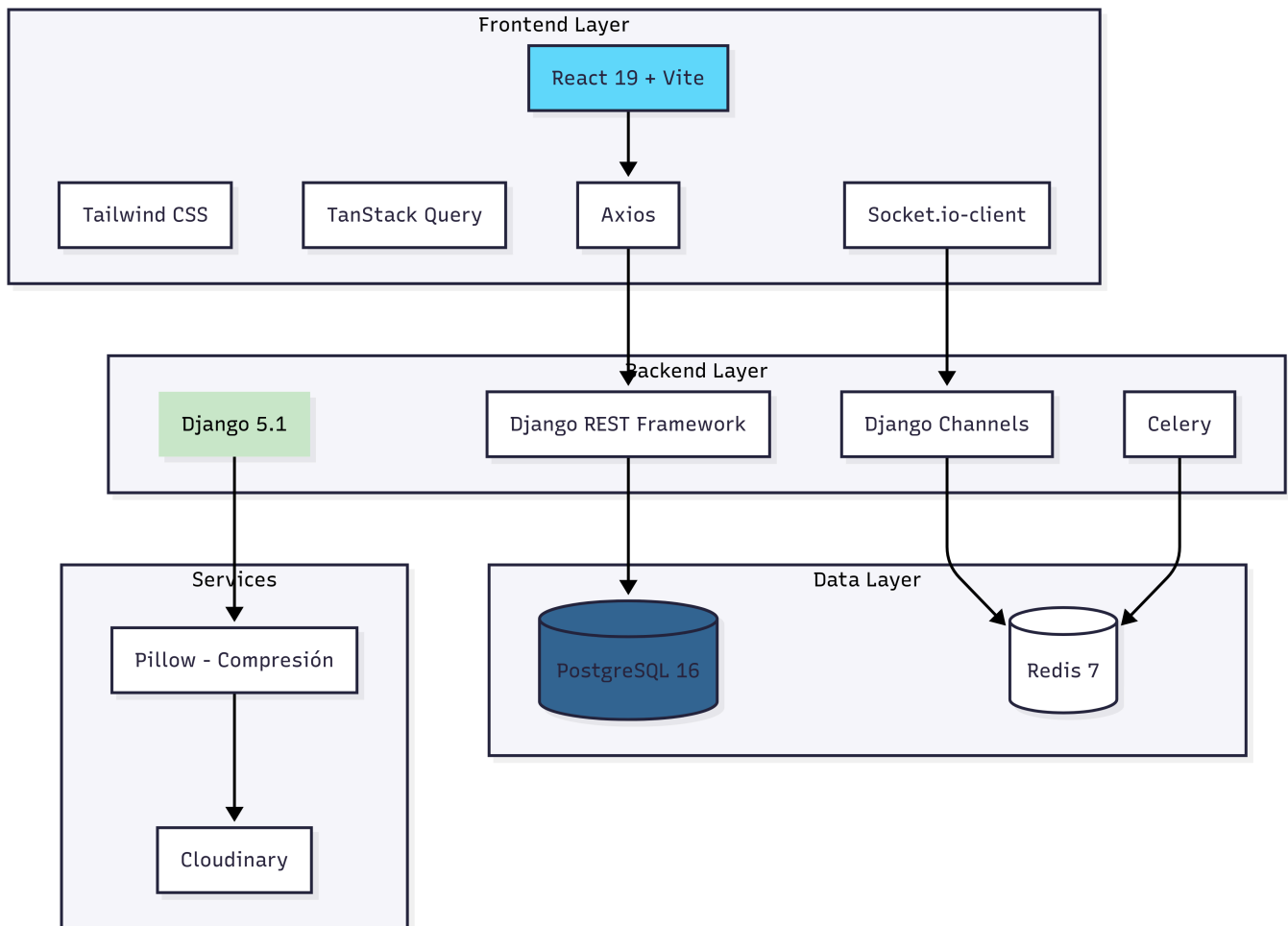
- Facilitar conexiones auténticas entre estudiantes de la universidad
- Crear un entorno seguro mediante verificación institucional (@unipamplona.edu.co)
- Desarrollar MVP funcional para 300 usuarios en fase piloto
- Aplicar habilidades técnicas demandadas en el mercado laboral 2025

Alcance Inicial

- **Usuarios objetivo:** 200-300 estudiantes (fase piloto)
- **Plataforma:** Web responsive (mobile-first)
- **Funcionalidades core:** Perfiles, swipe, matching, chat real-time
- **Timeline:** 6-8 semanas de desarrollo



Stack Tecnológico



Justificación por Capa

Frontend: React + Vite

- ✓ Experiencia del equipo: Ya conocen React
- ✓ Vite: 3-5x más rápido que Create React App
- ✓ Demanda laboral: React es #1 en frontend 2025
- ✓ Ecosistema maduro: Miles de librerías disponibles

Backend: Django + Django REST Framework

- ✓ Experiencia del equipo: Ya conocen Django/Python
- ✓ Batteries-included: ORM, admin panel, autenticación incluidos
- ✓ Madurez: 18+ años probado en producción
- ✓ Django admin: Panel de gestión sin código adicional

Base de Datos: PostgreSQL (Única)

- ✓ Aplicación relacional: Likes y matches son relaciones many-to-many
- ✓ Integridad ACID: Transacciones garantizadas
- ✓ JSONField nativo: Flexibilidad NoSQL cuando se necesita

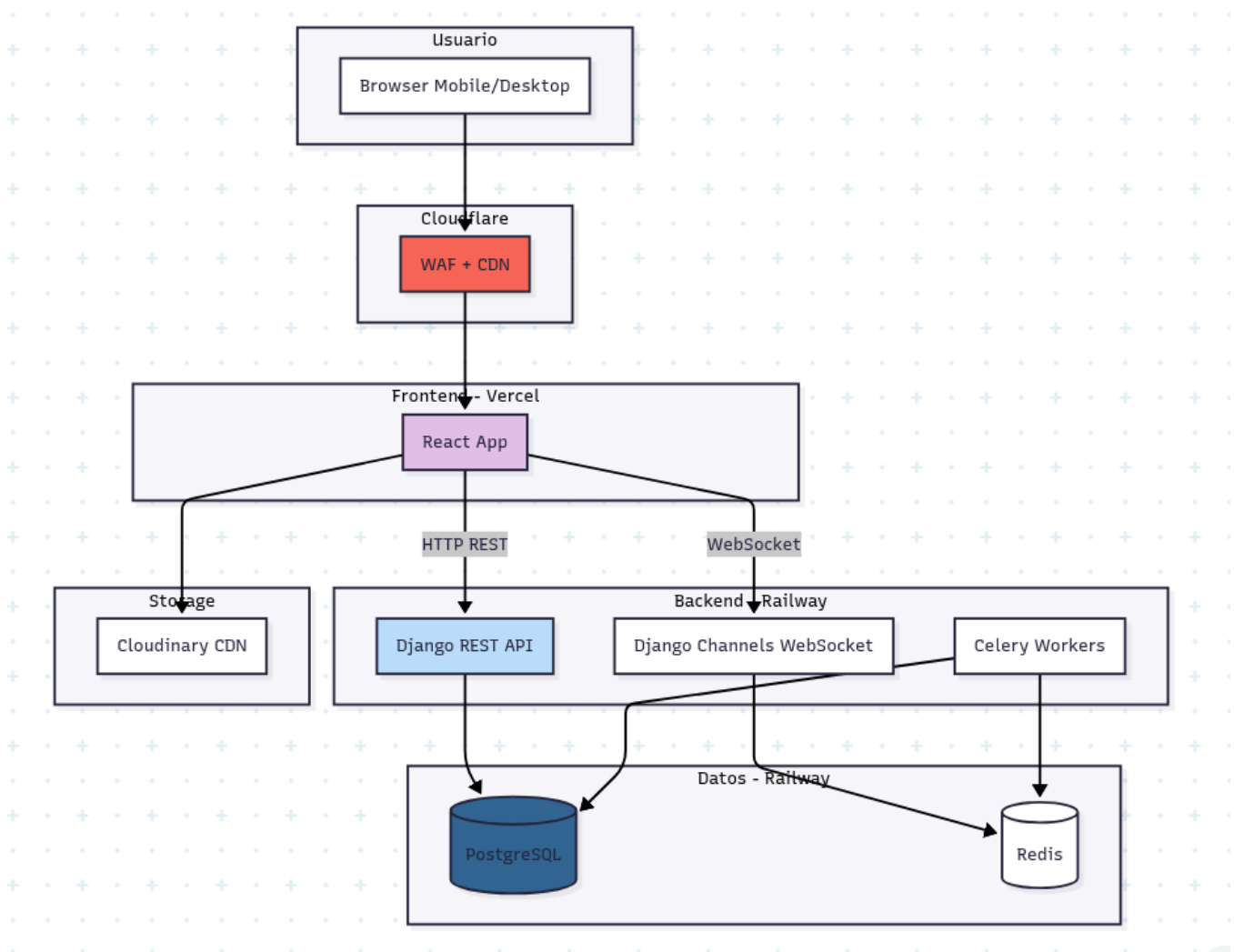
- ☒ Demanda laboral: Base de datos más solicitada

✓ **Decisión Crítica: Solo PostgreSQL** Se descartó MongoDB porque una dating app es inherentemente relacional. PostgreSQL con campos JSON ofrece lo mejor de ambos mundos sin la complejidad de mantener dos bases de datos debido a la decisión del profe Alejandro de manejar base de datos dinámica con sql y nosql, pero en el equipo no hay tanto manejo de mongo y es más complicado settarlo con el backend djando que vamos a trabajar.

Real-Time: Django Channels + Redis

- ☒ WebSockets nativos: Chat en tiempo real
- ☒ Integración Django: Auth y ORM funcionan naturalmente
- ☒ Redis como broker: Caché + Celery + Channels en uno

Arquitectura del Sistema



- ❗ Este diseño implementa una arquitectura de microservicios ligera con separación de responsabilidades, comunicación REST/WebSocket, procesamiento asíncrono con colas de tareas, y múltiples capas de caché para optimizar el rendimiento. La combinación de Django Channels, Celery y Redis permite manejar tanto operaciones síncronas como asíncronas, haciéndola ideal para aplicaciones que requieren actualizaciones en tiempo real y procesamiento de tareas en segundo plano.

Flujo de Datos Principal

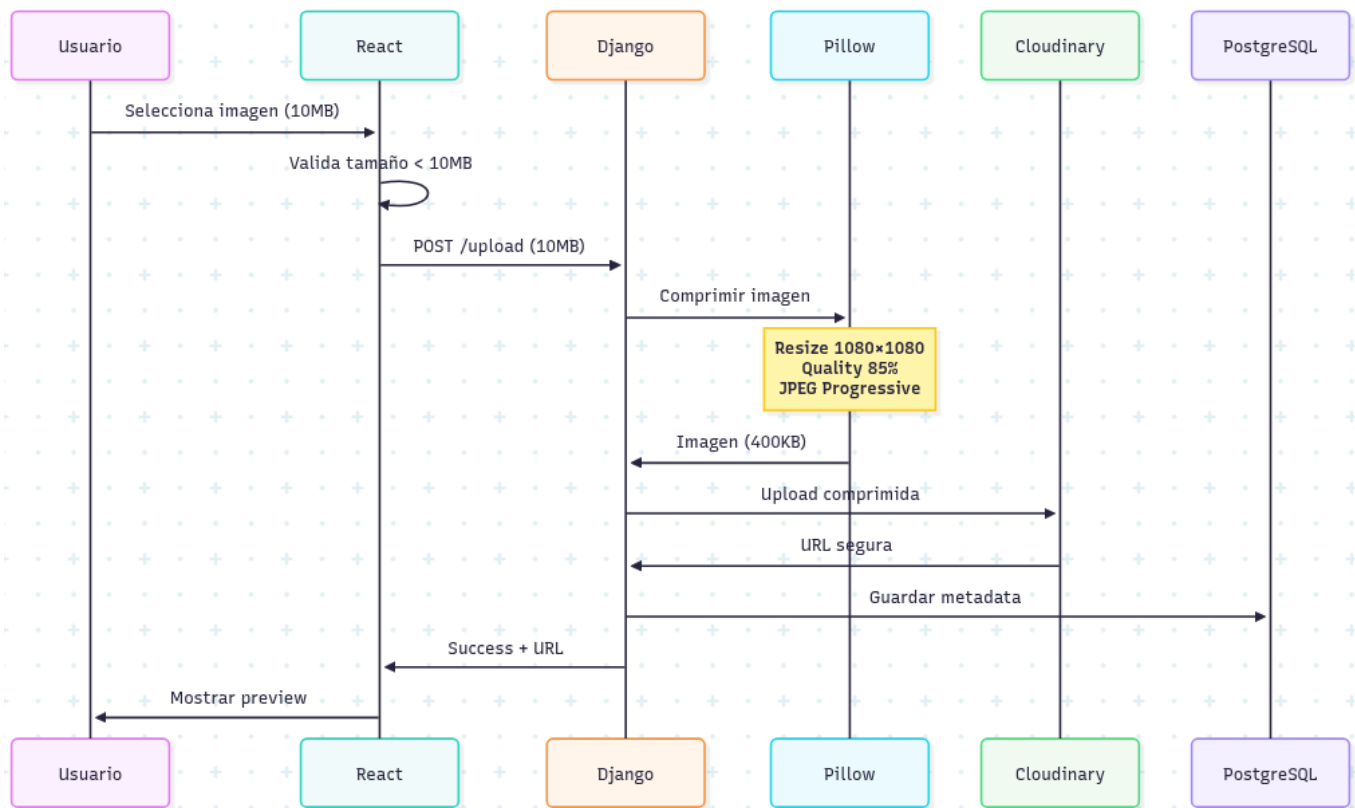
1. **Usuario** → Interactúa con React en navegador
2. **Cloudflare** → Protege contra ataques DDoS, caché estático
3. **React (Vercel)** → Renderiza UI, valida inputs
4. **Django API (Railway)** → Procesa lógica de negocio
5. **PostgreSQL** → Almacena datos estructurados
6. **Redis** → Caché, WebSocket rooms, Celery queue
7. **Cloudinary** → Almacena y entrega imágenes comprimidas



Gestión y Compresión de Imágenes

- ❗ **Corrección Importante** La nota original mencionaba "Squoosh API" pero Squoosh NO es una API de backend. Es una herramienta cliente-only que funciona en el navegador con WebAssembly.

Solución Adoptada: Pillow en Backend



Justificación Técnica

Aspecto	Squoosh Cliente	Pillow Backend	Decisión
Consistencia	Variable por dispositivo	Garantizada	✅ Backend
Complejidad	Alta (WebAssembly)	Baja (Python)	✅ Backend
Tiempo desarrollo	2-3 días	2-3 horas	✅ Backend
Experiencia usuario	Depende del celular	Siempre rápido	✅ Backend
Control calidad	Limitado	Total	✅ Backend
Ancho banda	Ahorra 10MB	Gasta 10MB	⚠️ Cliente

🔗 **Por Qué Backend Gana Aunque Pillow gasta más ancho de banda (10MB vs 400KB), para 300 usuarios MVP esto es solo 3GB/mes, equivalente al 3% del límite gratuito de Railway (100GB). La simplicidad y consistencia superan ampliamente el costo de bandwidth.**

Proceso de Compresión

1. Frontend (Validación):

- Verificar tamaño < 10MB
- Verificar dimensiones < 4000x4000px

- Mostrar mensaje de error si excede límites

2. Backend (Compresión Real):

- Pillow abre la imagen
- Corrige orientación EXIF (fotos de celular)
- Redimensiona a máximo 1080×1080px (mantiene aspect ratio)
- Comprime a JPEG quality 85%
- Genera JPEG progresivo para mejor UX
- Resultado: 10MB → 400KB (95% reducción)

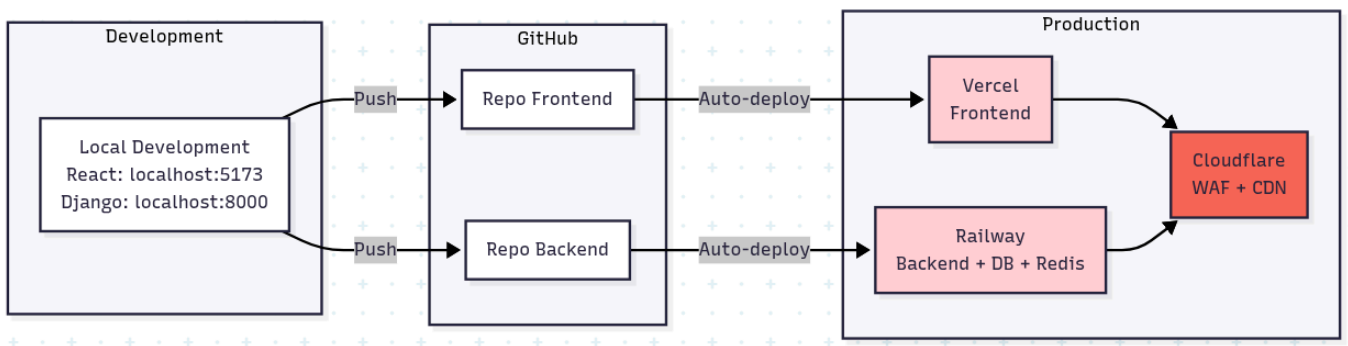
3. Storage (Cloudinary):

- Almacena imagen ya comprimida
- CDN global para entrega rápida
- WebP automático en navegadores compatibles
- Transformaciones on-the-fly si se necesita



Despliegue y Alojamiento

Arquitectura de Deployment



Plataformas Seleccionadas

Frontend: Vercel

- ☒ Free tier: 100GB bandwidth/mes
- ☒ Deploy automático desde GitHub
- ☒ CDN global incluido
- ☒ SSL/HTTPS automático
- ☒ Preview deployments por PR

Backend: Railway

- ☒ Free tier: \$5 crédito/mes
- ☒ PostgreSQL + Redis incluidos
- ☒ Deploy automático desde GitHub
- ☒ No duerme (vs Heroku/Render)
- ☒ Escalable cuando crezca

⚠ Alternativa Descartada Se desaconsejó usar servidores de la universidad por:

- Complejidad de configuración (Linux, red, firewall)
- Falta de equipo de soporte dedicado
- Problemas de conectividad externa
- Sin HTTPS automático

Protección: Cloudflare (Gratuito)

- ☒ DDoS protection
- ☒ Web Application Firewall (WAF)
- ☒ Rate limiting
- ☒ Bot protection
- ☒ Analytics de amenazas

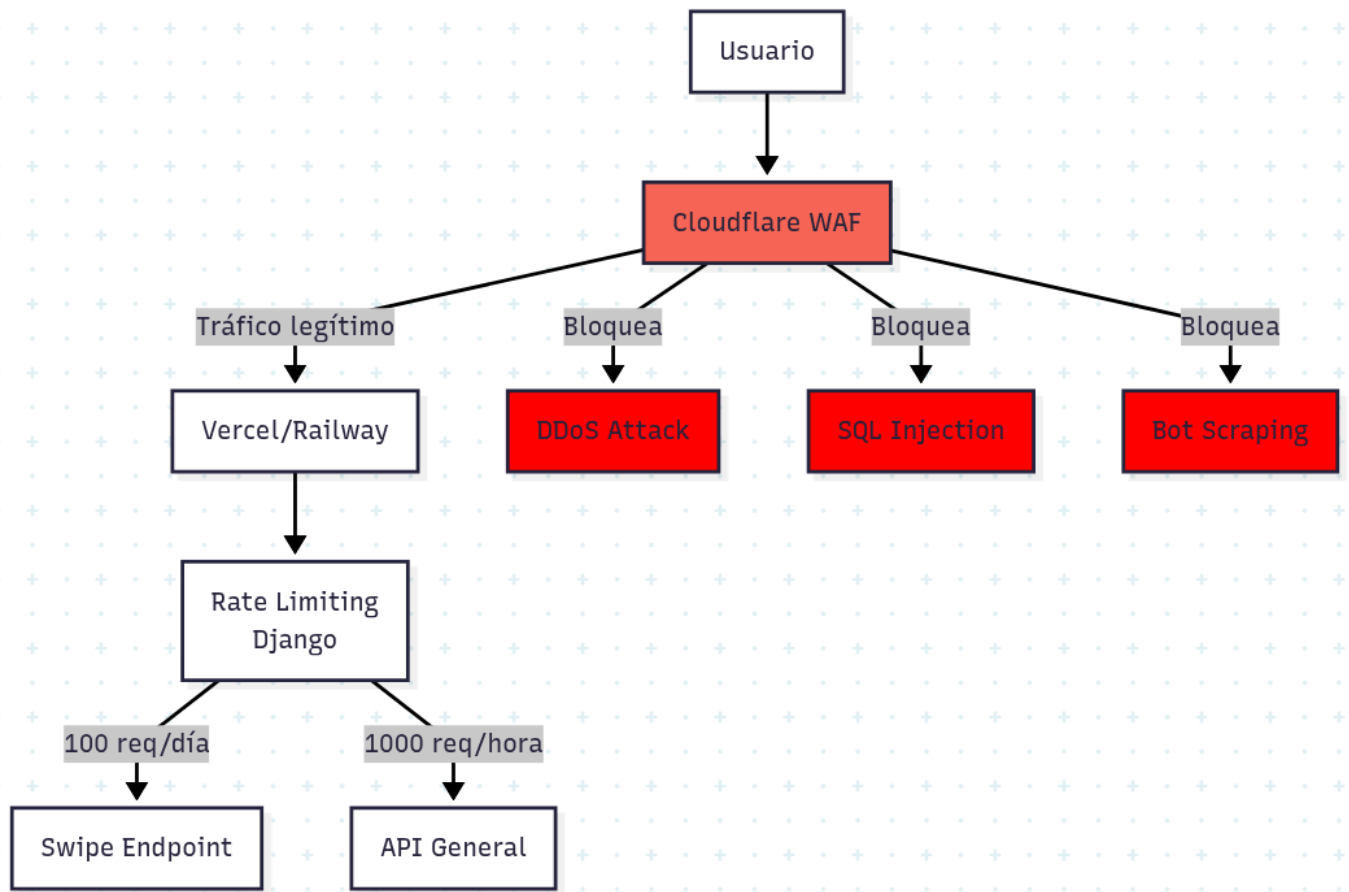
Costos Mensuales Estimados

Servicio	Plan	Costo
Vercel (Frontend)	Free	\$0
Railway (Backend)	Free tier	\$0-5
PostgreSQL	Incluido Railway	\$0
Redis	Incluido Railway	\$0
Cloudinary	Free tier	\$0
Cloudflare	Free	\$0
Total	-	\$0-5/mes

✓ **Viable para Proyecto Académico** El presupuesto total de \$0-5/mes hace que el proyecto sea completamente factible para un MVP universitario con 300 usuarios.

🔒 Seguridad y Escalabilidad

1. Protección contra Ataques



Capas de Protección

Capa 1: Cloudflare (Perimetral)

- DDoS protection automático
- Bot fight mode
- Rate limiting por IP global
- Bloqueo de países si es necesario

Capa 2: Django (Aplicación)

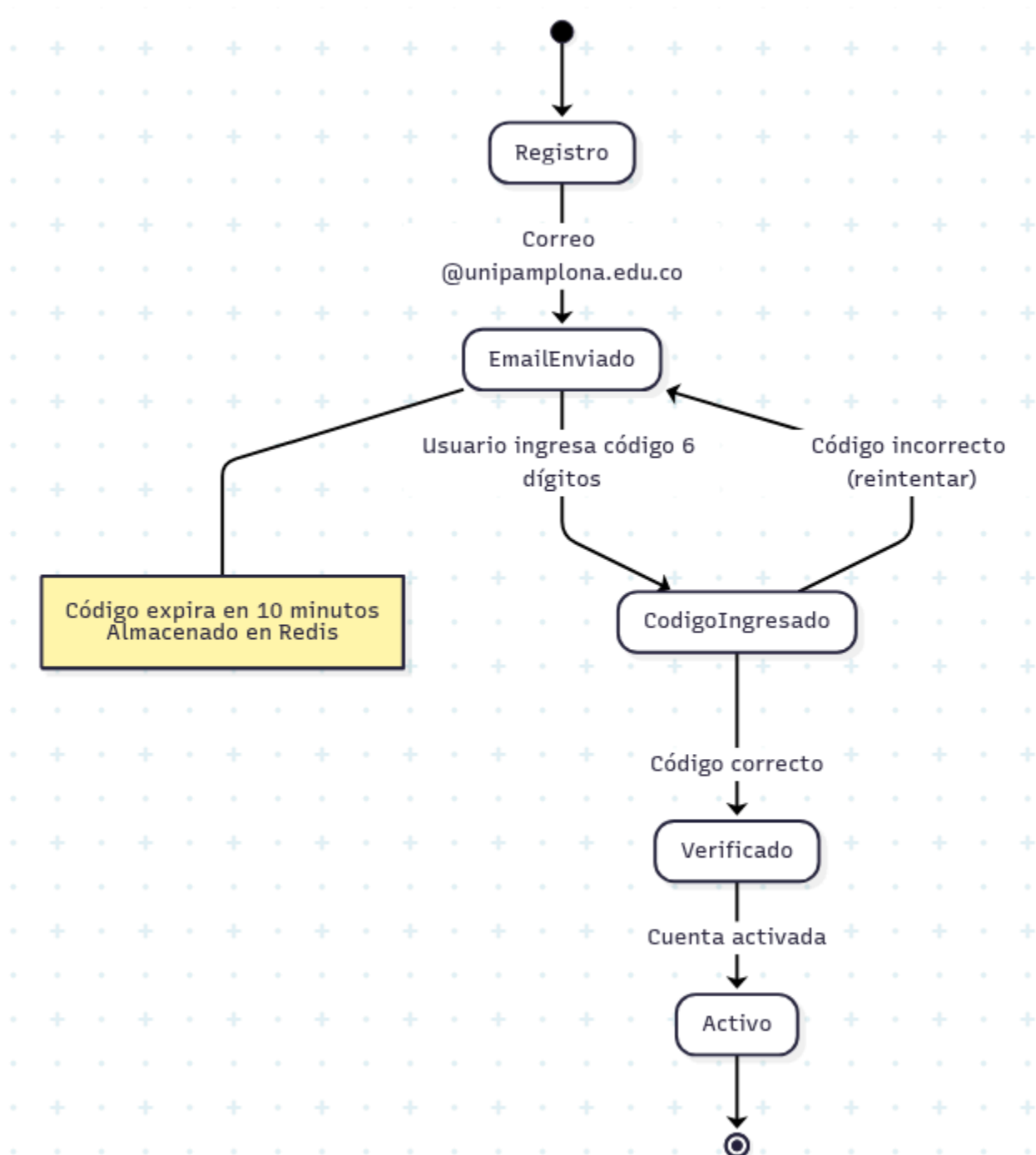
- Django REST Framework throttling
- Rate limiting específico por endpoint
- CSRF protection
- SQL injection prevention (ORM)
- XSS prevention (templates)

Capa 3: PostgreSQL (Datos)

- FOREIGN KEY constraints
- CHECK constraints
- Row-level security
- Conexiones SSL/TLS

⚠ **Importancia Crítica** Los ataques DDoS podrían generar costos elevados en Railway/Vercel. Cloudflare gratuito es esencial para evitar sorpresas en la factura.

2. Autenticación y Verificación



Flujo de Verificación

1. Usuario se registra con email @unipamplona.edu.co
2. Backend valida el dominio institucional
3. Genera código aleatorio de 6 dígitos
4. Almacena en Redis con TTL de 10 minutos
5. Envía email en texto plano con código
6. Usuario ingresa código en frontend
7. Backend verifica contra Redis
8. Activa cuenta si código coincide
9. Genera JWT tokens para sesión

🔗 **Simplicidad en Emails** No complicarse con plantillas HTML elaboradas. Un email de texto plano con el código es suficiente para la versión inicial. Se puede mejorar después.

Características de Seguridad

- ☒ Solo correos institucionales válidos
- ☒ Código expira en 10 minutos
- ☒ Un código por usuario
- ☒ Máximo 3 intentos de código incorrecto
- ☒ Rate limiting en endpoint de verificación
- ☒ JWT tokens con refresh automático

3. Manejo de Datos y Privacidad

Cumplimiento Legal (Colombia)

🔗 **Ley 1581 de 2012 - Habeas Data** Se debe implementar una política de tratamiento de datos para cumplir con la legislación colombiana de protección de datos personales.

Datos Recolectados:

- Email institucional (obligatorio)
- Nombre, edad, género (obligatorio)
- Fotografías de perfil (obligatorio)
- Intereses y preferencias (opcional)
- Historial de likes/matches
- Mensajes de chat

Derechos del Usuario:

- Conocer qué datos se almacenan
- Actualizar/rectificar datos
- Eliminar cuenta y datos (derecho al olvido)
- Exportar todos sus datos (portabilidad)
- Revocar consentimiento

Implementación Técnica:

- Endpoint `/api/users/export-data/` para exportar
- Endpoint `/api/users/delete-account/` para eliminación
- 30 días de gracia antes de borrado definitivo
- Logs de acceso a datos personales

Versionamiento y Flujo de Trabajo

Estrategia de Repositorios

❗ Decisión: Dos Repositorios Separados Se dividirá el proyecto en dos repos independientes para facilitar el trabajo en paralelo y deployment separado.

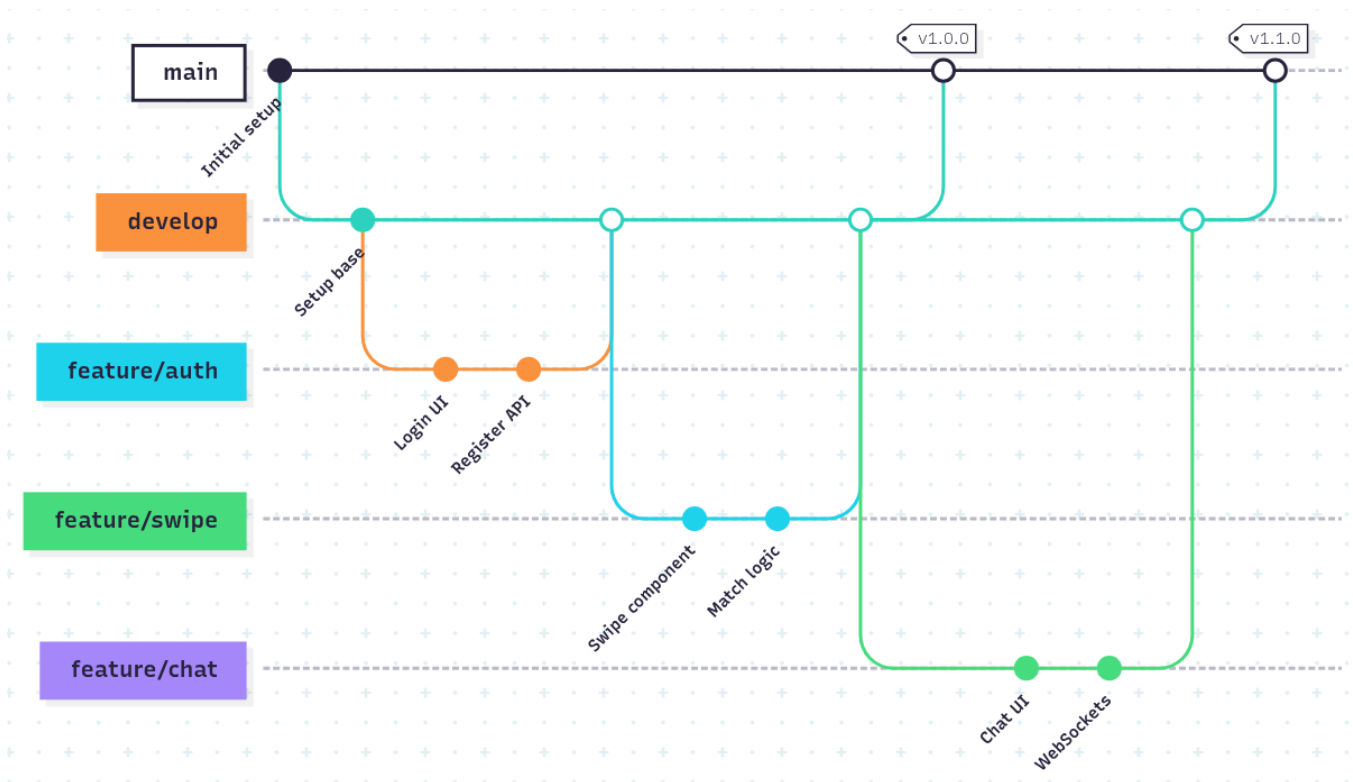
```

GitHub Organization: cUPido-App
|
├── cupido-frontend/           (React + Vite)
|   ├── main                  (Producción)
|   ├── develop               (Desarrollo)
|   └── feature/*              (Features individuales)
└── cupido-backend/           (Django + DRF)
    ├── main                  (Producción)
    ├── develop               (Desarrollo)
    └── feature/*              (Features individuales)
  
```

Justificación:

- Tecnologías y dependencias diferentes
- Equipos frontend/backend pueden trabajar en paralelo
- Deployment independiente (Vercel vs Railway)
- Git history más limpio por proyecto

Git Flow Simplificado



Branches Principales

- **main**: Producción estable, siempre deployable
- **develop**: Integración continua, features completas
- **feature/***: Desarrollo de features individuales

Flujo de Trabajo

1. Crear feature branch desde develop
2. Desarrollar feature con commits descriptivos
3. Push a GitHub y crear Pull Request
4. Code review por al menos 1 compañero
5. Merge a develop después de aprobación
6. Deploy a staging automático desde develop
7. Merge a main cuando se valide en staging
8. Deploy a producción automático desde main

Convenciones de Commits

```
feat(scope): descripción corta
fix(scope): descripción del bug
docs(scope): cambio en documentación
style(scope): formato, espacios
refactor(scope): refactorización
test(scope): tests
```

```
chore(scope): mantenimiento
```

Ejemplos:

```
feat(auth): add email verification
```

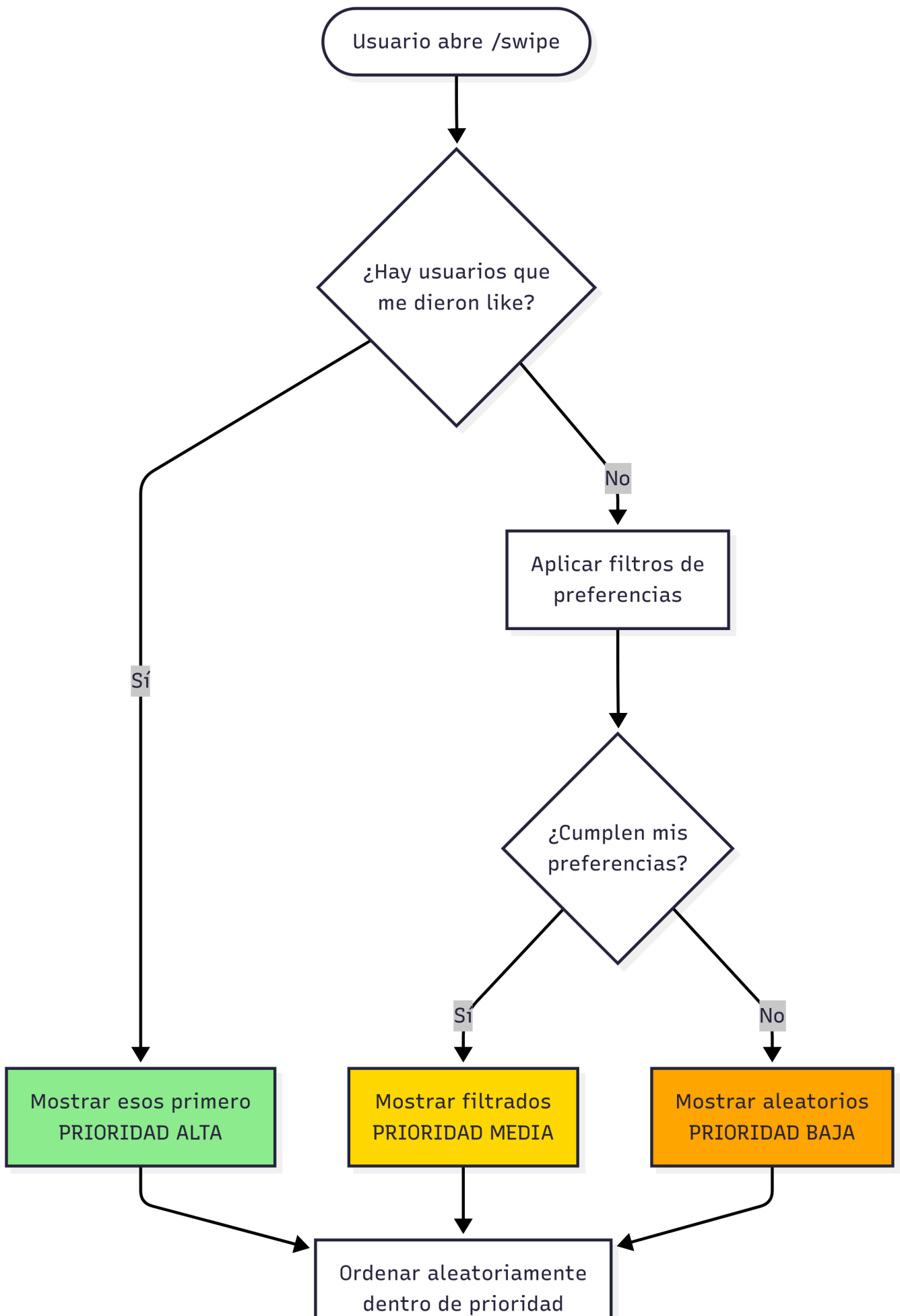
```
fix(swipe): fix match detection bug
```

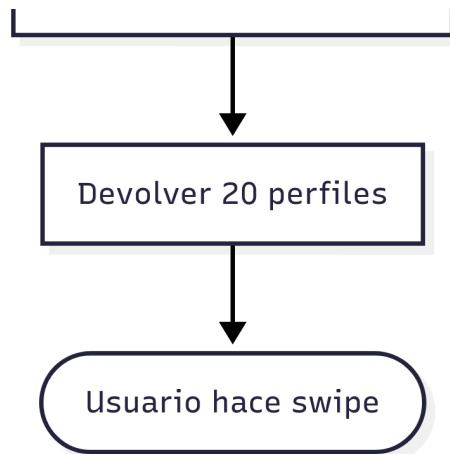
```
docs(readme): update setup instructions
```

✨ Funcionalidades Clave y Algoritmos

Algoritmo de Matching Simplificado

🔗 **Recomendación del Equipo NO utilizar Inteligencia Artificial en la fase inicial por su complejidad y costo computacional. Algoritmo simple basado en priorización de likes mutuos.**





Lógica del Algoritmo

Prioridad 1 (Alta) - Likes Recibidos:

- Usuarios que ya me dieron like aparecen primero
- Facilita matches rápidos
- Aumenta engagement

Prioridad 2 (Media) - Preferencias:

- Filtros de edad (ej. 18-28 años)
- Filtros de género
- No se han visto antes

Prioridad 3 (Baja) - Aleatorios:

- Cuando se agotan las opciones anteriores
- Perfiles completamente aleatorios
- Evita mostrar siempre a las mismas personas

Datos a Considerar

Criterios de Filtrado:

- Rango de edad (configurable por usuario)
- Géneros de interés (configurable)
- Verificación completada
- Perfil activo (no eliminado)
- No bloqueado por el usuario

NO Considerar Inicialmente:

- Distancia geográfica (no se usa geolocalización)
- Intereses en común (puede añadirse después)

- Compatibilidad algorítmica (futuro)

Implementación de Gustos/Preferencias

✓ **Decisión: Tags Predefinidos** Usar opciones predefinidas (checkboxes, dropdowns) en lugar de texto libre. Simplifica búsquedas y normalización.

Categorías de Intereses Sugeridas:

- **Música:** Rock, Pop, Reggaeton, Electrónica, Salsa, etc.
- **Deportes:** Fútbol, Baloncesto, Ciclismo, Gym, Yoga, etc.
- **Hobbies:** Lectura, Gaming, Cocina, Fotografía, Viajes, etc.
- **Académico:** Ingeniería, Artes, Ciencias, Medicina, etc.

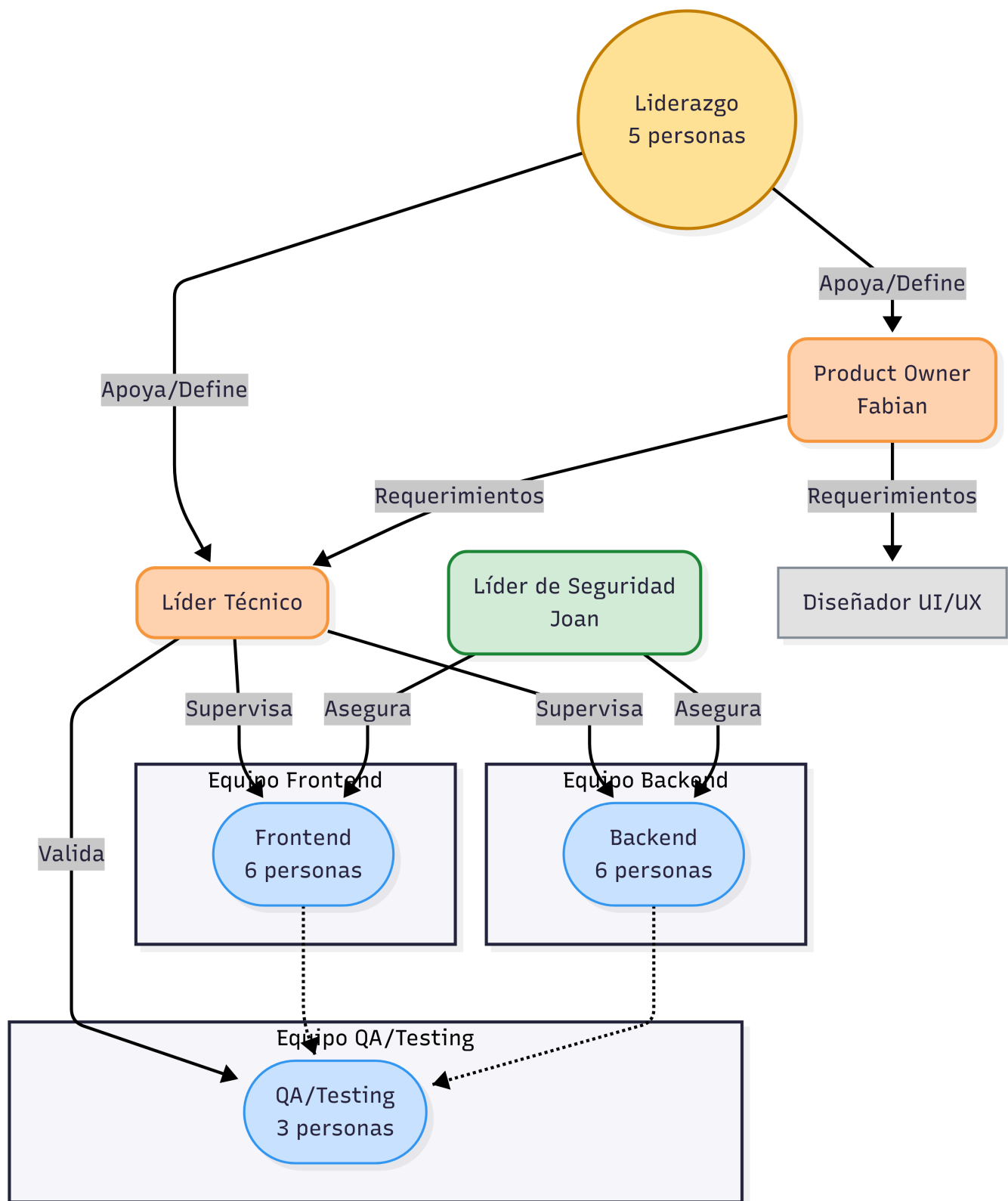
Formato en Base de Datos:

```
PostgreSQL - Campo interests (Array)
{
  "interests": [
    "música-rock",
    "deportes-fútbol",
    "hobbies-gaming",
    "académico-ingeniería"
  ]
}
```



Equipo y Roles

Estructura del Equipo (~30 personas)



Roles Identificados

Equipos:

- **Frontend Team:** React, UI/UX, interacciones
- **Backend Team:** Django, APIs, lógica de negocio
- **QA/Testing:** Pruebas, bugs, calidad

Buenas Prácticas de Trabajo

🔗 Trabajo en Equipo Efectivo

- **Confiar en el trabajo de otros:** No permitir que una persona rehaga todo
- **Code reviews obligatorios:** Todos los PRs deben ser revisados
- **Pair programming:** Para features complejas o críticas
- **Daily standups:** 15 minutos para sincronizar equipo
- **Retrospectivas:** Cada 2 semanas para mejorar procesos

⚠️ **Prevenir Sobrecarga** Es fundamental que los líderes técnicos deleguen efectivamente y no terminen rehaciendo el trabajo de otros. Esto fomenta el aprendizaje y evita burnout.



Consejos y Buenas Prácticas

🔗 **Enfoque Backend-First** Todo lo que se pueda procesar en el backend (compresión, validaciones complejas, lógica de negocio) sin recargar la interacción de red del cliente, es preferible hacerlo ahí.

🔗 **Selección de Tecnologías** No elegir librerías o frameworks por moda o novedad, sino porque resuelven un problema específico del proyecto de manera eficiente. Usar lo que el equipo ya conoce para agilizar el desarrollo.

🔗 **Prueba Piloto y Lanzamiento Controlado** Antes de un lanzamiento masivo, realizar una prueba piloto con un grupo pequeño y controlado de usuarios (10-20 personas) para detectar errores y recibir feedback.

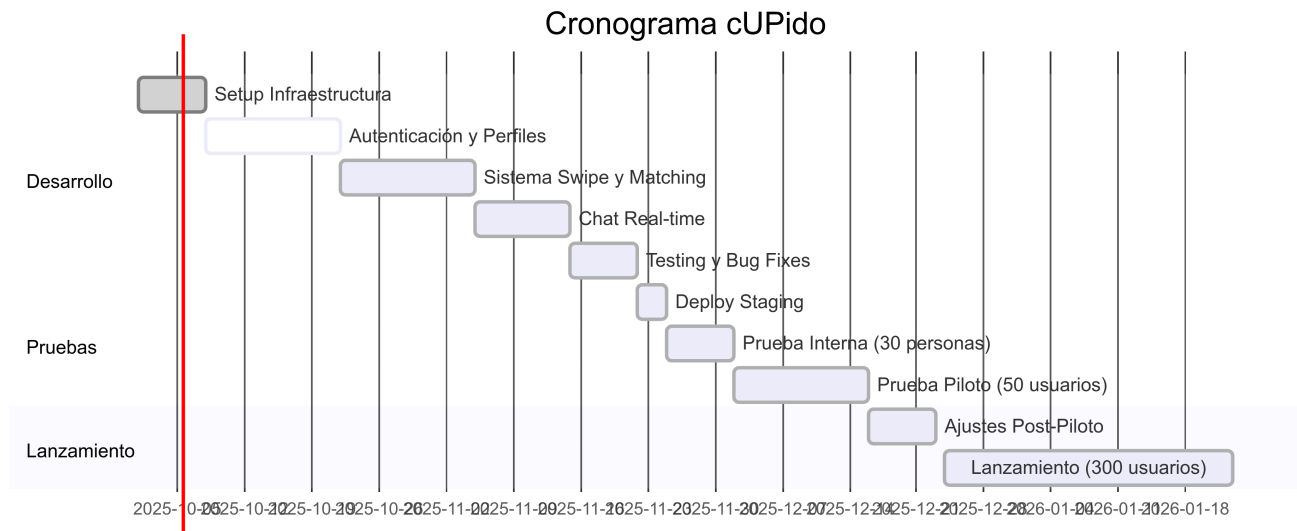
🔗 **Control de Costos y Límites** Implementar en el backend un límite de inscripciones (200-300 usuarios). Cuando se alcance, el sistema debe bloquear nuevos registros automáticamente. Esto es crucial para evitar costos inesperados en Railway/Vercel.

🔗 **Priorizar Funcionalidad sobre Estética** Enfocarse primero en que la funcionalidad principal sea sólida y funcione correctamente. Las mejoras visuales y estéticas se pueden realizar después. "Gástenme tiempo en la funcionalidad".



Plan de Lanzamiento

Fases del Proyecto



Checklist Pre-Lanzamiento

Backend Completo:

- ☐ Todas las APIs documentadas
- ☐ Tests unitarios > 80% coverage
- ☐ Rate limiting configurado
- ☐ Límite de usuarios implementado (300)
- ☐ Email verification funcionando
- ☐ WebSockets estables
- ☐ Logs centralizados

Frontend Completo:

- ☐ Responsive mobile + desktop
- ☐ Animaciones de swipe fluidas
- ☐ Chat en tiempo real funcional
- ☐ Manejo de errores robusto
- ☐ Loading states
- ☐ Tests E2E de flujos críticos

Infraestructura Lista:

- ☐ SSL/TLS en todos los endpoints

- ☐ Cloudflare configurado
- ☐ Monitoring de costos
- ☐ Alertas de downtime
- ☐ Backups automáticos DB

Legal y Documentación:

- ☐ Política de privacidad publicada
- ☒ Términos y condiciones
- ☐ Consentimiento datos
- ☐ README con setup
- ☐ Documentación API



Contexto Adicional del Proyecto

Oportunidades Académicas

📌 Inscripción como Proyecto de Investigación Se mencionó la posibilidad de inscribir cUPido como proyecto en un grupo de investigación universitario para obtener puntos académicos.

Áreas de Investigación Potenciales:

- Algoritmos de matching en redes sociales
- Comportamiento de usuarios en dating apps universitarias
- Seguridad y privacidad en aplicaciones sociales
- Arquitecturas full-stack modernas

Evaluación del Proyecto

Sistema de Calificación Interno:

- Implementar sistema de 1 a 5 estrellas dentro de la app
- Medir satisfacción del usuario
- Obtener ranking de features más valoradas
- Feedback cualitativo vía encuestas

Métricas de Éxito:

- Daily Active Users (DAU) > 30%
- Match rate > 10%

- Message rate > 40% de matches
- Retention D7 > 20%
- NPS (Net Promoter Score) > 30



Comparativa Técnica: Decisiones Clave

PostgreSQL vs MongoDB

Criterio	PostgreSQL	MongoDB	Decisión
Relaciones Many-to-Many	★★★★★ Nativo	★★ Complicado	✓ PostgreSQL
Integridad Referencial	★★★★★ FOREIGN KEYS	★★ Manual	✓ PostgreSQL
Queries Complejas	★★★★★ SQL + JOINS	★★ Agregaciones	✓ PostgreSQL
Flexibilidad Schema	★★★★★ JSONField	★★★★★ Schema-less	✓ PostgreSQL suficiente
Transacciones ACID	★★★★★ Nativo robusto	★★★★ Desde v4.0	✓ PostgreSQL
Demanda Laboral 2025	★★★★★ Muy alta	★★★★ Alta	✓ PostgreSQL
Curva Aprendizaje	★★★★★ SQL estándar	★★★★ NoSQL mindset	✓ PostgreSQL

✓ **Conclusión Técnica PostgreSQL es objetivamente superior para una dating app relacional. MongoDB no aporta ventajas reales y añade complejidad innecesaria.**

Compresión: Backend vs Cliente

Aspecto	Pillow Backend	Squoosh Cliente	Decisión
Consistencia	★★★★★ Garantizada	★★ Variable	✓ Backend
Complejidad	★★★★★ Baja	★★ Alta	✓ Backend
Tiempo Desarrollo	★★★★★ 1 día	★★ 2 días	✓ Backend
Experiencia Usuario	★★★★★ Consistente	★★★★ Variable	✓ Backend

Aspecto	Pillow Backend	Squoosh Cliente	Decisión
Ancho de Banda	★★★★ Mayor	★★★★★ Mínimo	⚠ Cliente mejor
Control Calidad	★★★★★ Total	★★ Limitado	✅ Backend
Apropiado para MVP	★★★★★ Perfecto	★★★ Overkill	✅ Backend

✓ **Conclusión Técnica Pillow en backend es superior porque la consistencia y simplicidad superan el costo de 3GB extras de bandwidth mensual para 300 usuarios.**



Recursos y Referencias

Documentación Oficial

- [DJANGO DOCUMENTATION](#)
- [REACT DOCUMENTATION](#)
- [POSTGRESQL DOCS](#)
- [DJANGO REST FRAMEWORK](#)
- [DJANGO CHANNELS](#)

Deployment Platforms

- [VERCEL](#)
- [RAILWAY](#)
- [CLOUDFLARE](#)
- [CLOUDINARY](#)



Próximos Pasos Inmediatos

Semana 1: Setup Inicial

- ☐ Crear organización GitHub "cUPido-App"
- ☐ Crear repos `cupido-frontend` y `cupido-backend`
- ☐ Setup Django proyecto base
- ☐ Setup React + Vite proyecto base

- ☐ Configurar Railway (PostgreSQL + Redis)
- ☒ ~~Configurar Vercel~~
- ☒ ~~Primera deploy de "Hello World"~~

Semana 2-3: Autenticación

- ☐ Django REST Framework + JWT
- ☐ Endpoint registro con validación @unipamplona.edu.co
- ☐ Sistema de verificación por email
- ☐ Login/logout
- ☐ React: Páginas de auth
- ☐ Axios interceptors para JWT

Semana 4-5: Perfiles y Swipe

- ☐ Modelo de Profile en Django
- ☐ Upload y compresión de fotos con Pillow
- ☐ Cloudinary integration
- ☐ API de sugerencias de perfiles
- ☐ React: Página de swipe con animaciones
- ☐ Algoritmo de matching

Semana 6-7: Chat Real-time

- ☐ Django Channels setup
- ☐ WebSocket consumers
- ☐ Redis como channel layer
- ☐ React: Componente de chat
- ☐ Socket.io-client integration
- ☐ Notificaciones de nuevos mensajes

Semana 8: Testing y Deployment

- ☐ Tests unitarios backend
 - ☐ Tests E2E frontend
 - ☐ Deploy a producción
 - ☐ Prueba piloto con 10-20 usuarios
 - ☐ Ajustes y bug fixes
-



Resumen Final

Conclusión cUPido es un proyecto técnicamente viable con stack moderno (Django + React + PostgreSQL), arquitectura simple pero profesional, y costos controlados (\$0-5/mes). Las decisiones técnicas están fundamentadas en la experiencia del equipo, demanda laboral y simplicidad de desarrollo para un MVP universitario exitoso.

Última actualización: 2025-10-05

Estado: En desarrollo

Próxima revisión: Post-prueba piloto

Referencias

- [1] [HTTPS://WWW.REDDIT.COM/R/DJANGO/COMMENTS/1CEUQ2B/DJANGO_ARCHITECTURE_DESIGN/](https://www.reddit.com/r/Django/comments/1CEUQ2B/Django_architecture_design/)
- [2] [HTTPS://WWW.YOUTUBE.COM/WATCH?V=38XWpyEK8IY](https://www.youtube.com/watch?v=38XWpyEK8IY)
- [3] [HTTPS://WWW.DIGITALOCEAN.COM/COMMUNITY/TUTORIALS/COMO-CREAR-UNA-APLICACION-WEB-MODERNA-PARA-GESTIONAR-LA-INFORMACION-DE-CLIENTES-CON-DJANGO-Y-REACT-ON-UBUNTU-18-04-ES](https://www.digitalocean.com/community/tutorials/como-crear-una-aplicacion-web-moderna-para-gestionar-la-informacion-de-clientes-con-django-y-react-on-ubuntu-18-04-es)
- [4] [HTTPS://DEV.TO/MAXWELLNEWAGE/DIARIO-DE-PYTHON-15-COMBINANDO-DJANGO-Y-REACT-13JC](https://dev.to/maxwellnewage/diario-de-python-15-combinando-django-y-react-13jc)
- [5] [HTTPS://PLATZI.COM/BLOG/REACT-JS-EN-SERVIDOR-CON-DJANGO/](https://platzi.com/blog/react-js-en-servidor-con-django/)
- [6] [HTTPS://WWW.YOUTUBE.COM/WATCH?V=KILUIyF91AA](https://www.youtube.com/watch?v=KILUIyF91AA)
- [7] [HTTPS://ES.LINKEDIN.COM/POSTS/ALBERRDEV_DESPLEGAR-APLICACI%C3%B3N-FULL-STACK-CON-REACT-ACTIVITY-7221543713385156608-GsIC](https://es.linkedin.com/posts/alberrdev_desplegar-aplicaci%C3%B3n-full-stack-con-react-activity-7221543713385156608-GsIC)
- [8] [HTTPS://HACKERNOON.COM/LANG/ES/ARQUITECTURA-DJANGO-SAAS-INQUILINO-UNICO-VS-MULTI-INQUILINO-QUE-ES-ADECUADO-PARA-USTED](https://hackernoon.com/lang/es/arquitectura-django-saas-inquilino-unico-vs-multi-inquilino-que-es-adecuado-para-usted)