Estado Actual del Proyecto - Agente de Ventas AI

> **Grupo 3** - Sistema Agéntico de Comercio Inteligente
>
> Documento Ejecutivo - Febrero 2026

Resumen Ejecutivo

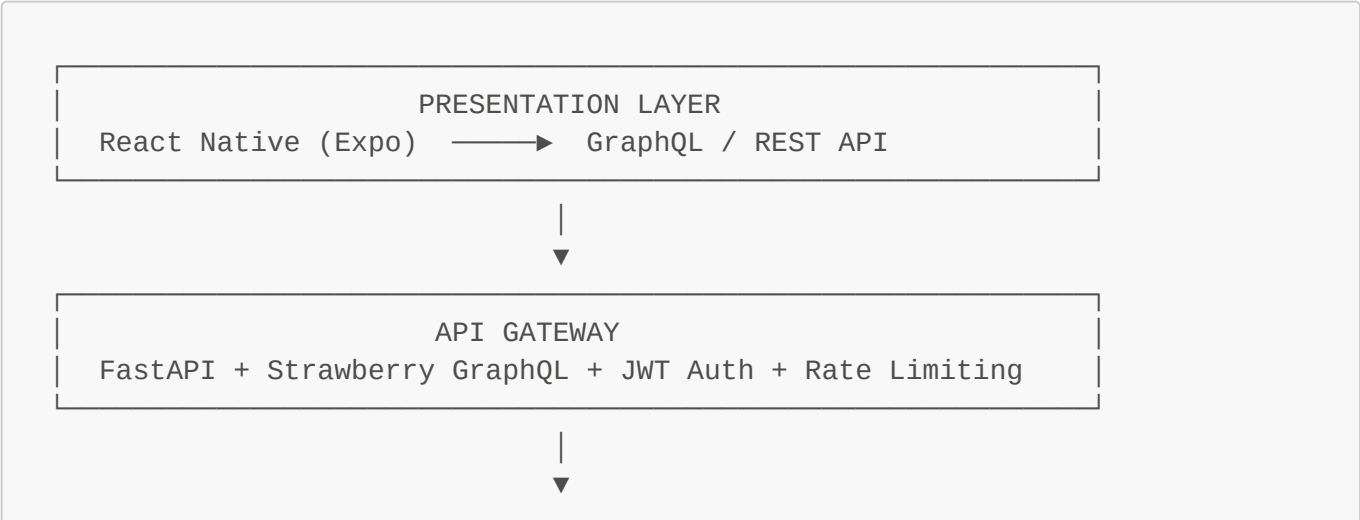
El proyecto implementa un **Asistente de Ventas Inteligente (Alex)** para comercio electrónico de calzado deportivo. El sistema está completamente funcional en su núcleo, con arquitectura multi-agente, persistencia en PostgreSQL, caché/sesiones en Redis, y RAG con ChromaDB.

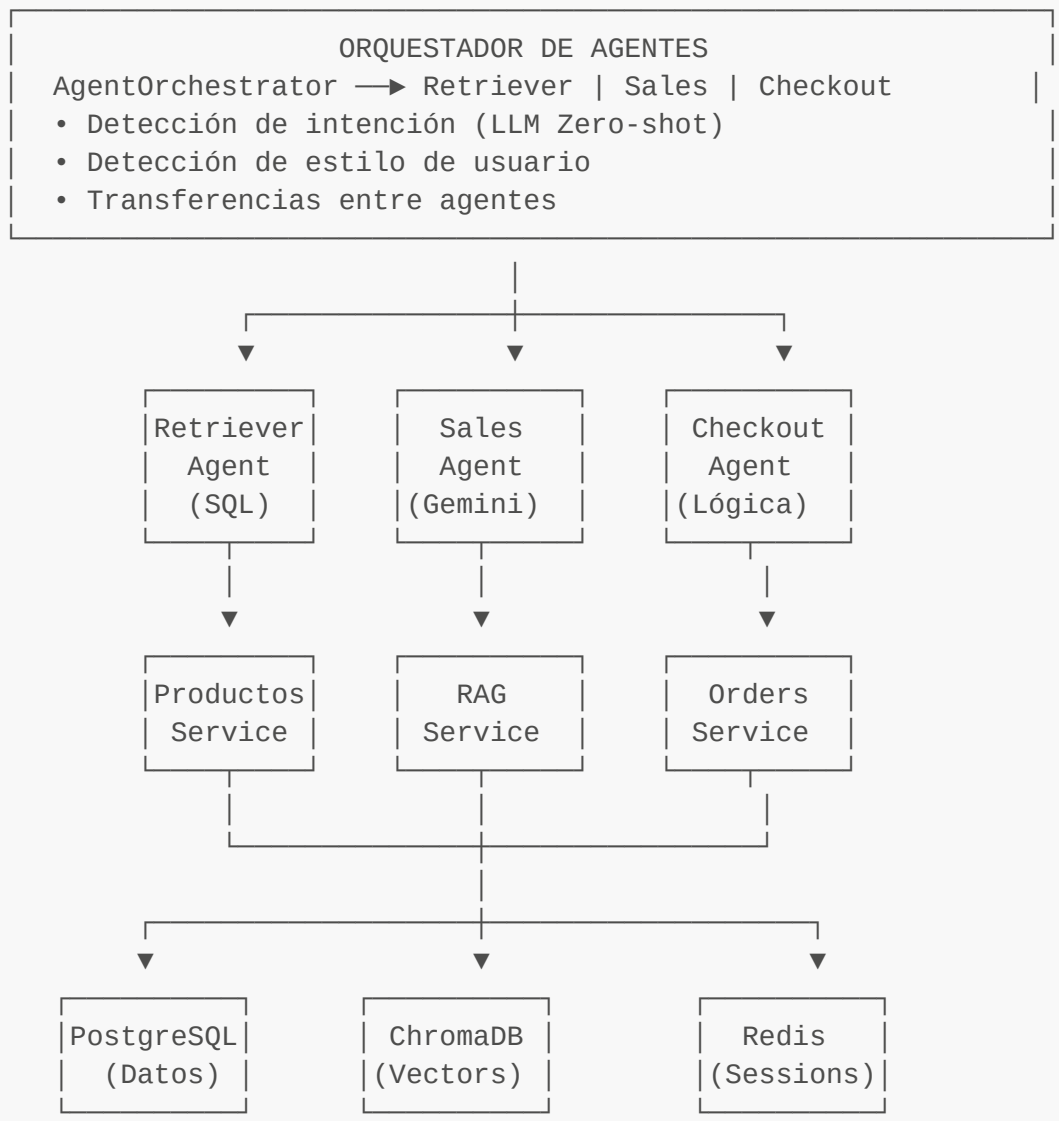
Estado General: **OPERATIVO**

Componente	Estado	Notas
Backend API	✅ Funcional	FastAPI + GraphQL operativo
Autenticación JWT	✅ Funcional	Login/registro con bcrypt
Multi-Agente	✅ Funcional	3 agentes coordinados
Base de Datos	✅ Funcional	PostgreSQL con modelos completos
RAG (ChromaDB)	✅ Funcional	Embeddings con Vertex AI
Sesiones Redis	✅ Funcional	Con fallback a memoria
Agente 2 (Visión)	⚠️ Parcial	Cliente integrado, requiere servicio externo
Agente 1 (Inventario)	⏳ Pendiente	Slot reservado para integración
Tests	⚠️ Básicos	Estructura lista, cobertura limitada

Arquitectura General

Stack Tecnológico





Dependencias Externas

Servicio	Uso	Estado
Google Vertex AI	LLM (Gemini 1.5 Flash) + Embeddings	✅ Requerido
PostgreSQL 15	Datos transaccionales	✅ Requerido
Redis	Sesiones y caché	⚠️ Opcional (fallback a memoria)
Agente 2 (Puerto 5000)	Reconocimiento visual SIFT	⚠️ Opcional

Sistema Multi-Agente

Arquitectura de Agentes

El sistema implementa un **orquestador central** que coordina 3 agentes especializados:

1. RetrieverAgent (Buscador)

- **Función**: Búsqueda rápida en base de datos SQL

- **Técnica**: SQLAlchemy async con filtros por palabras clave
- **Trigger**: Intención `search` o palabras clave de búsqueda
- **No usa LLM**: Solo lógica y SQL
- **Salida**: Lista de productos disponibles

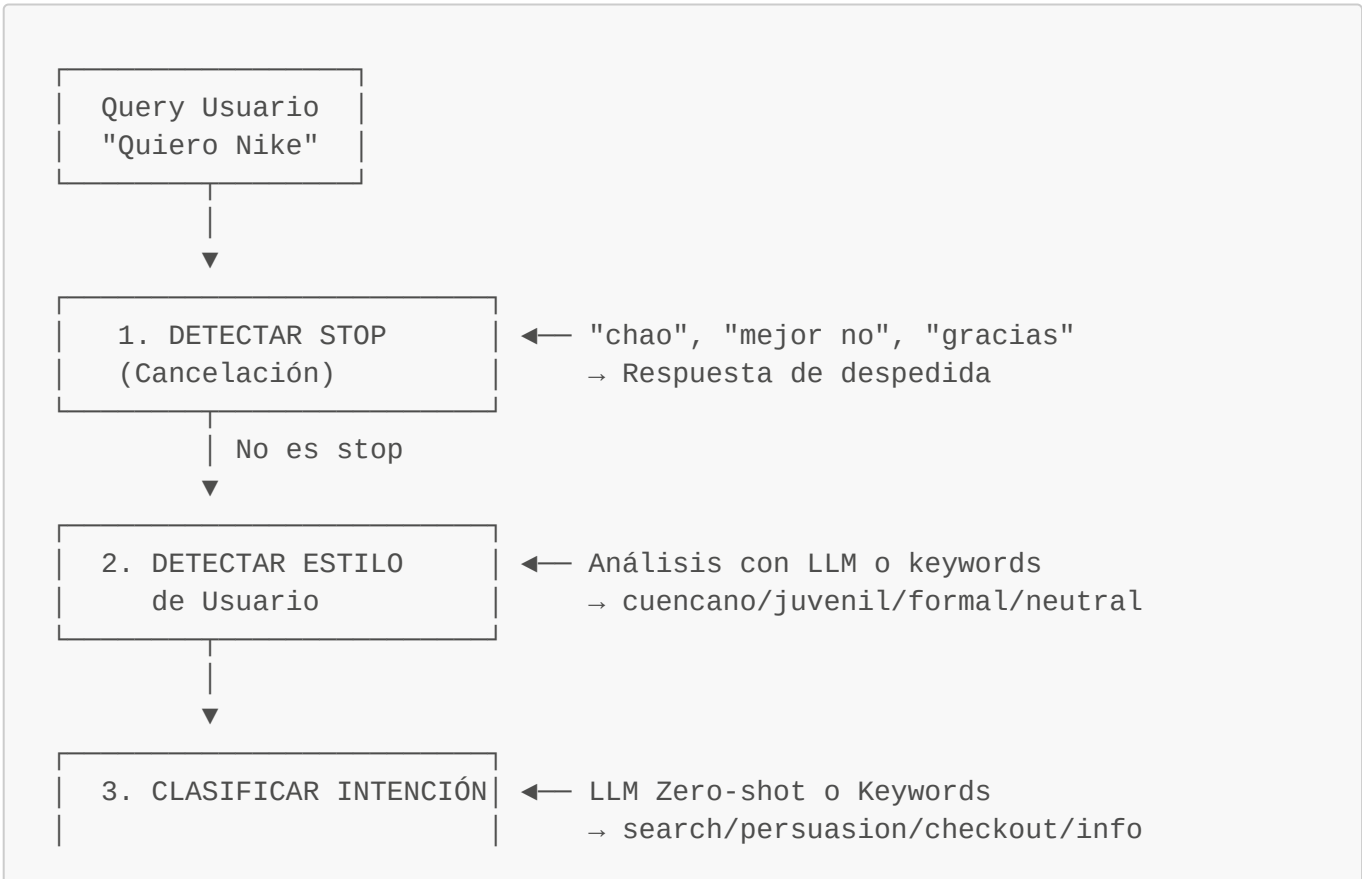
2. SalesAgent (Vendedor "Alex")

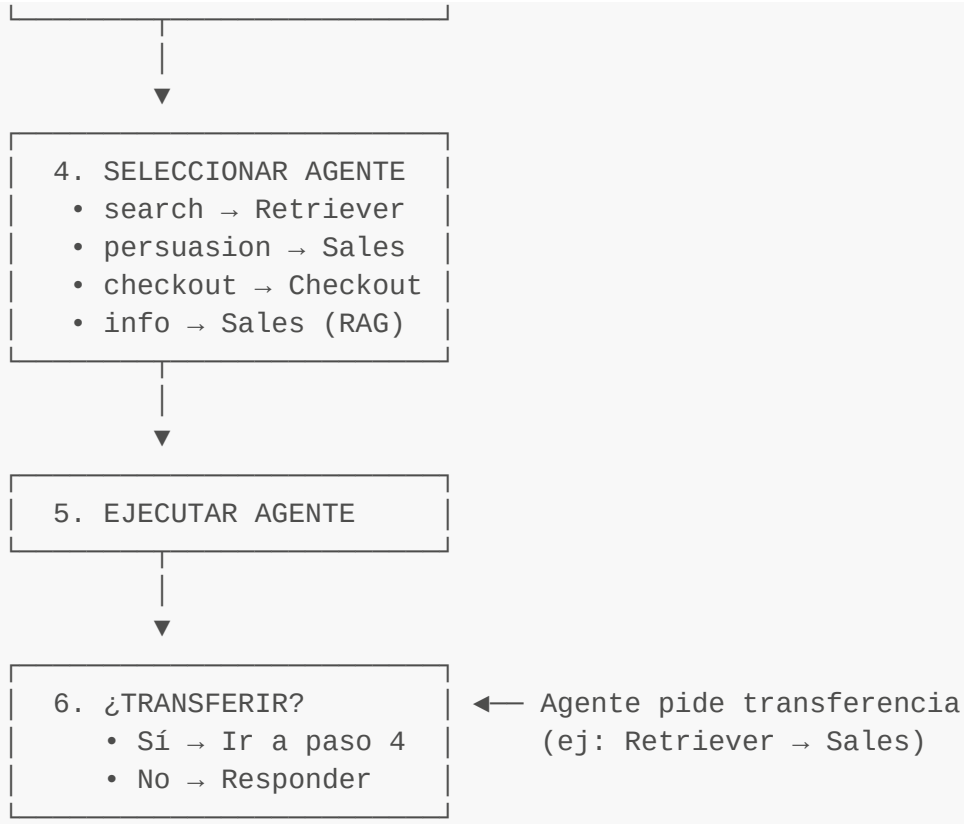
- **Función**: Persuasión, recomendaciones, manejo de objeciones
- **Técnica**: Gemini 1.5 Flash con prompts adaptativos
- **Trigger**: Intención `persuasion`, `info`, o contexto de venta
- **Características**:
  - Adaptación de tono según estilo detectado (cuencano/juvenil/formal/neutral)
  - Integración con RAG para FAQs
  - Reconocimiento de imágenes (Agente 2)
  - Slot filling (memoria de contexto)
  - Anti-alucinación (solo menciona productos de la lista)

3. CheckoutAgent (Cajero)

- **Función**: Cierre de transacciones
- **Técnica**: Lógica transaccional pura (sin LLM)
- **Trigger**: Intención `checkout` o flujo de confirmación
- **Flujo**:
  1. Iniciar → 2. Confirmar producto → 3. Dirección → 4. Pago → 5. Confirmación
- **Validaciones**: Stock en tiempo real, usuario autenticado

Flujo de Decisión de Queries





Detección de Intención (LLM Zero-shot)

El orquestador usa **Gemini** para clasificar intenciones:

Prompt al LLM:

```
Clasifica la intención en UNA de estas categorías:
1. search: Buscar/explorar productos
2. persuasion: Dudas, objeciones, recomendaciones
3. checkout: Comprar/confirmar pedido
4. info: Preguntas generales (horarios, políticas)

Responde SOLO con JSON:
{
  "intent": "search|persuasion|checkout|info",
  "confidence": 0.0-1.0,
  "reasoning": "explicación"
}
```

Fallback: Si el LLM falla o timeout (>5s), usa keywords predefinidas.

Detección de Estilo de Usuario

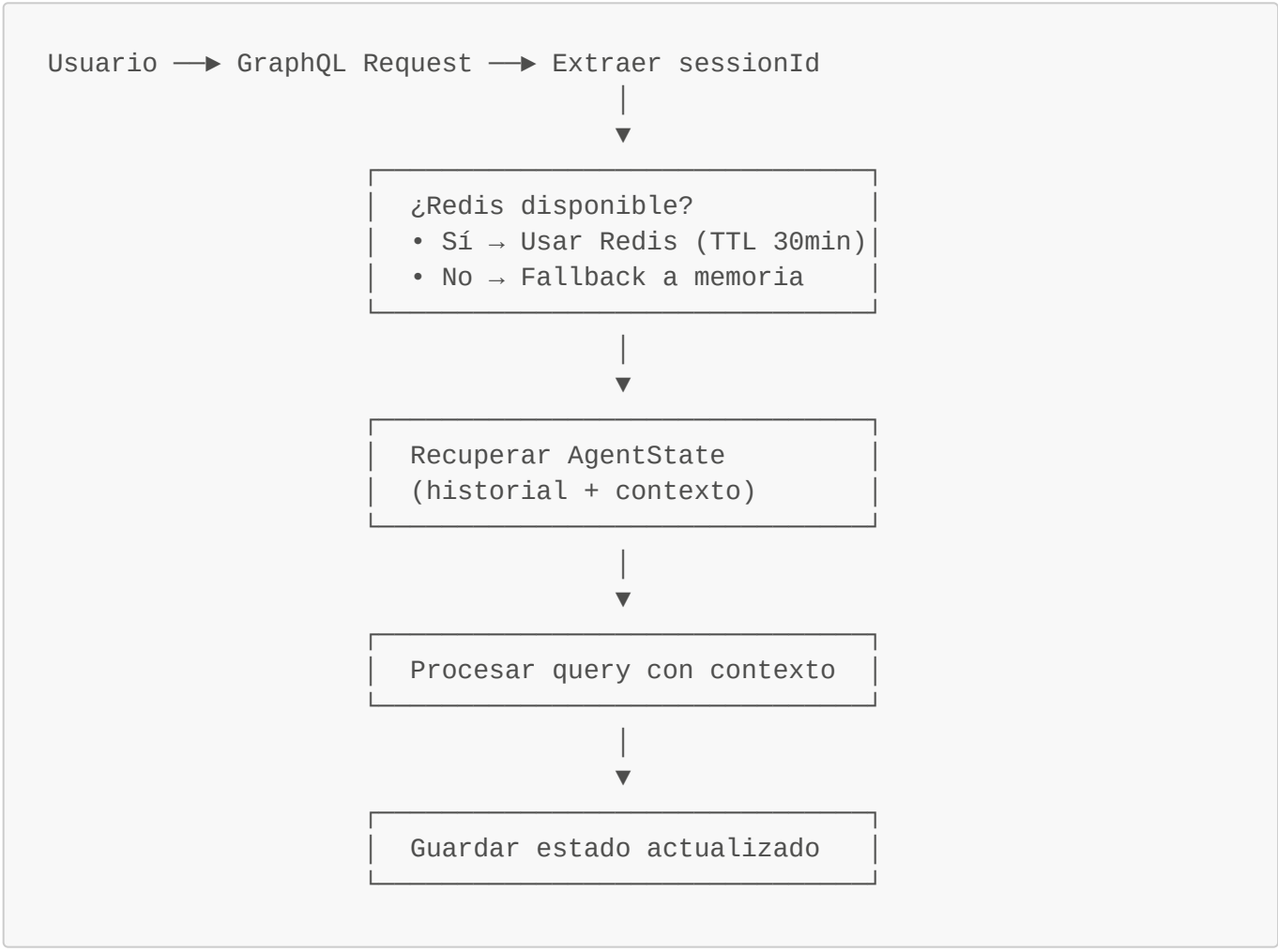
4 estilos soportados para adaptar el tono de Alex:

Estilo	Marcadores	Ejemplo

Estilo	Marcadores	Ejemplo
cuencano	"ayayay", "ve", "full", "chevere", "lindo"	"Ayayay que lindo ve, busco unos Nike full buenos"
juvenil	"che", "bro", "tipo", "re", "copado"	"Che bro, mostrame algo copado tipo para correr"
formal	"usted", "señor", "por favor", "quisiera"	"Buenos días, quisiera consultar por zapatillas"
neutral	Ninguno marcado	"Hola, busco zapatillas Nike"

💾 Manejo de Sesiones

Arquitectura de Sesiones



Estructura del Estado (AgentState)

```
AgentState {
  # Conversación
  user_query: str                    # Query actual
  conversation_history: []           # Historial de mensajes

  # Contexto del usuario
  user_style: "cuencano|juvenil|formal|neutral"
  detected_intent: "search|persuasion|checkout|info"
```

```
  # Estado de búsqueda
  search_results: []                # Productos encontrados
  selected_products: []             # Productos en carrito

  # Slot Filling
  conversation_slots: {             # Info ya obtenida
    product_name: "Nike",
    size: "42",
    color: "negro",
    activity_type: "correr"
  }

  # Checkout
  checkout_stage: "confirm|address|payment|complete"
  shipping_address: str

  # Imagen (Agente 2)
  detected_product_from_image: str
  image_recognition_confidence: float

  # Metadata
  session_id: str
  user_id: str                      # UUID usuario autenticado
  timestamp: datetime
}
```

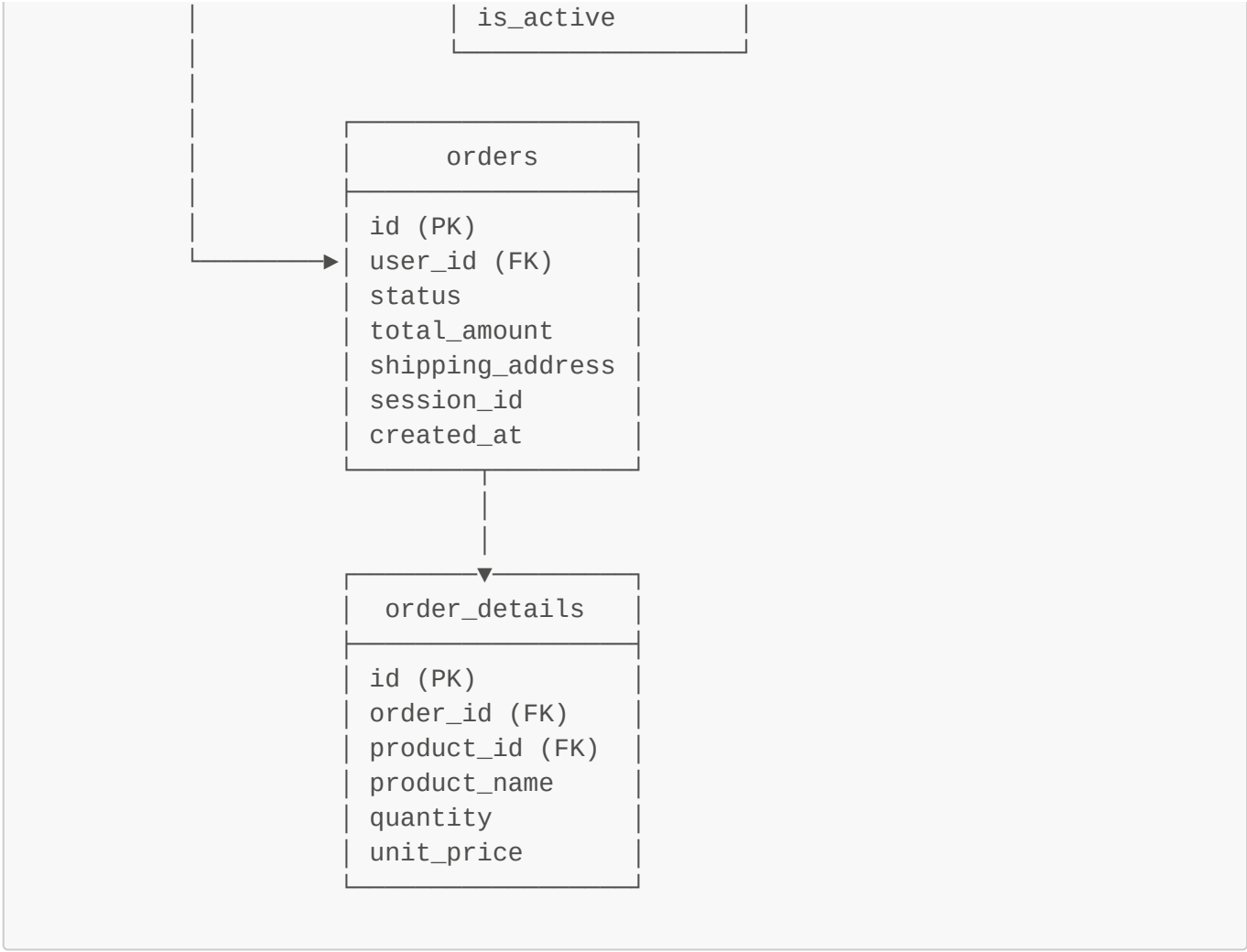
Persistencia

Modo	TTL	Caso de uso
Redis	30 minutos	Producción, escalable
Memoria	Hasta reinicio	Desarrollo, fallback

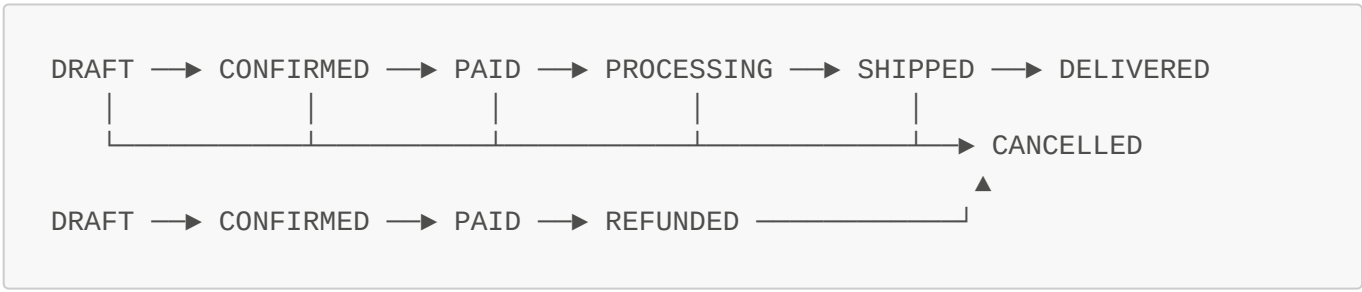
Base de Datos

Esquema Relacional

```
┌──────────────────────┐       ┌──────────────────────┐
│      users           │       │   product_stocks     │
├──────────────────────┤       ├──────────────────────┤
│ id (PK)              │       │ id (PK)              │
│ username (UQ)        │       │ product_name         │
│ email (UQ)           │       │ product_sku          │
│ password_hash        │       │ quantity_avail       │
│ full_name            │       │ unit_cost            │
│ role (1|2)           │       │ warehouse_loc        │
│ is_active            │       │ shelf_location       │
│ created_at           │       │ batch_number         │
└──────────┬───────────┘       │ stock_status         │
```



Estados de Orden (OrderStatus)



Modelos Implementados

Modelo	Descripción	Relaciones
`User`	Usuarios del sistema	1:N con Order
`ProductStock`	Inventario de productos	1:N con OrderDetail
`Order`	Cabecera de pedidos	N:1 con User, 1:N con OrderDetail
`OrderDetail`	Líneas de pedido	N:1 con Order, N:1 con ProductStock

🔌 API y Endpoints

REST Endpoints (Auth)

Endpoint	Método	Auth	Rate Limit	Descripción
`/auth/login`	POST	No	5/min	Login con username/email
`/auth/rate-limit-status`	GET	No	-	Info de rate limits

GraphQL Schema

Queries

```
# Productos
listProducts(limit: Int): [ProductStockType!]!
getProductById(id: UUID!): ProductStockType

# Usuarios
getCurrentUser: UserType
getUserById(id: UUID!): UserType

# Órdenes
getOrderById(id: UUID!): OrderType
getMyOrders(limit: Int, offset: Int): [OrderSummaryType!]!
getRecentOrders(limit: Int, statusFilter: String): [OrderSummaryType!]!

# Chat
semanticSearch(query: String!, sessionId: String): SemanticSearchResponse!
```

Mutations

```
# Usuarios
createUser(input: CreateUserInput!): AuthResponse!
updateUser(input: UpdateUserInput!): UserType!
changePassword(input: ChangePasswordInput!): Boolean!

# Órdenes
createOrder(input: CreateOrderInput!): CreateOrderResponse!
cancelOrder(orderId: UUID!, reason: String): CreateOrderResponse!

# Reconocimiento
recognizeProductImage(image: Upload!): ProductRecognitionResponse!
```

Autenticación

```
          | (HS256 + Secret)|
          └─────────┬─────────┘
                    |
          ┌─────────▼─────────┐
          | ¿Válido?          |
          | • Sí → Procesar   |
          | • No → 401        |
          └───────────────────┘
```

Claims del JWT:

- `id`: UUID del usuario
- `username`: Nombre de usuario
- `email`: Correo
- `role`: 1=Admin, 2=Cliente
- `exp`: Expiración (30 minutos)

🐳 Infraestructura

Docker Compose

El archivo `docker-compose.yml` define:

Servicio	Puerto	Descripción
`postgres`	5432	Base de datos principal
`redis`	6379	Caché y sesiones
`backend`	8000	API FastAPI
`agent2`	5000	Servicio de reconocimiento (externo)

Variables de Entorno Requeridas

```
# Base de datos
PG_URL=postgresql+asyncpg://user:pass@localhost:5432/dbname

# Google Cloud
GOOGLE_CLOUD_PROJECT=mi-proyecto-gcp
GOOGLE_APPLICATION_CREDENTIALS=/ruta/a/credenciales.json

# Agente 2
AGENT2_URL=http://localhost:5000
AGENT2_ENABLED=true

# Redis (opcional)
REDIS_URL=redis://localhost:6379
```

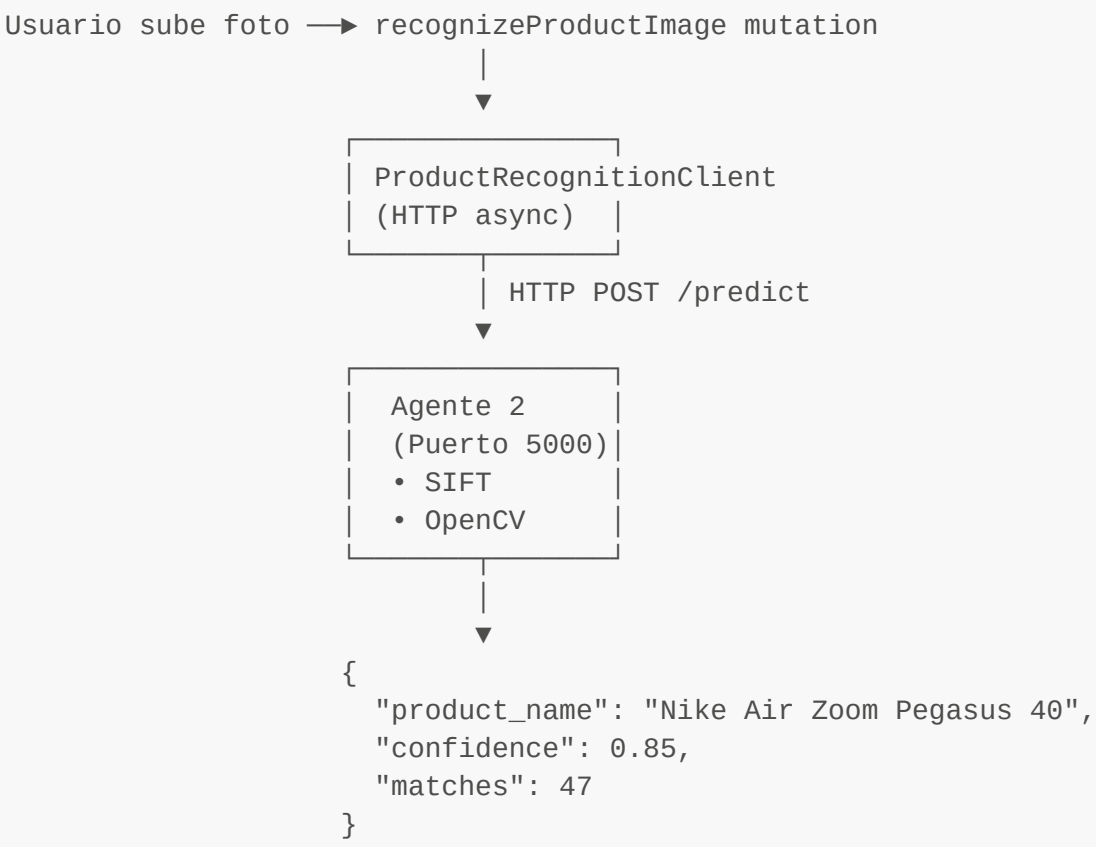
Sin Docker (Desarrollo Local)

```
# 1. PostgreSQL local
sudo systemctl start postgresql

# 2. Redis local (opcional)
redis-server

# 3. Backend
uv run python -m backend.main

# 4. Agente 2 (si se tiene)
# cd ../agente2 && python app.py
```

Integraciones

Agente 2 - Reconocimiento Visual (INTEGRADO)

Estado: Cliente integrado, requiere servicio externo



Endpoints del Agente 2:

- `POST /predict` - Reconocer producto en imagen
- `POST /register` - Registrar nuevo producto
- `GET /health` - Health check

Agente 1 - Gestión de Inventario (PENDIENTE)

Estado: Slot reservado, pendiente de integración

Funcionalidad esperada:

- Registro de productos desde fotos (OCR + Vision)
- Pipeline: Capture → OCR → Normalizer → DB Writer → Verifier
- Endpoints esperados:
  - `POST /register-product` - Crear producto desde imágenes
  - `POST /extract-from-image` - Extraer datos de etiquetas

Integración propuesta:

```
# backend/tools/agent1_inventory_client.py (pendiente crear)
class InventoryAgentClient:
    async def register_from_images(self, images: List[bytes]) ->
ProductData:
        # Llamar a Agente 1
        pass
```

Estructura de Carpetas

```
backend/
├── agents/                 # Sistema multi-agente
│   ├── base.py            # Clase base BaseAgent
│   ├── orchestrator.py    # AgentOrchestrator (orquestador)
│   ├── retriever_agent.py # Agente de búsqueda SQL
│   ├── sales_agent.py     # Agente vendedor (LLM)
│   └── checkout_agent.py  # Agente de checkout
├── api/
│   ├── endPoints/auth/    # REST endpoints (login)
│   └── graphql/           # Schema GraphQL
│       ├── queries.py     # Queries (listProducts, semanticSearch)
│       ├── mutations.py   # Mutations (createOrder, etc)
│       └── types.py       # Tipos Strawberry
├── config/                # Configuración
│   ├── settings.py        # Variables de entorno
│   ├── security/          # JWT, bcrypt
│   └── redis_config.py    # Config Redis
├── database/
│   ├── models/            # Modelos SQLAlchemy
│   │   ├── user_model.py
│   │   ├── product_stock.py
│   │   ├── order.py
│   │   └── order_detail.py
│   ├── session.py         # Sesiones DB
│   └── connection.py      # Conexión async
├── domain/                # Schemas Pydantic
```

```
│   ├── agent_schemas.py   # AgentState, AgentResponse
│   ├── order_schemas.py
│   └── product_schemas.py
├── llm/                   # Proveedor LLM
│   └── provider.py        # Gemini/Vertex AI
├── services/              # Lógica de negocio
│   ├── search_service.py  # Punto entrada búsquedas
│   ├── product_service.py # CRUD productos
│   ├── order_service.py   # CRUD órdenes
│   ├── user_service.py    # CRUD usuarios
│   ├── rag_service.py     # ChromaDB + Embeddings
│   └── session_service.py # Redis sessions
├── tools/                 # Clientes externos
│   └── agent2_recognition_client.py
├── tests/                 # Tests
│   ├── integration/
│   └── unit/
├── container.py           # Inyección de dependencias (AIOInject)
└── main.py               # Punto de entrada
```

Tests y Calidad

Cobertura Actual

Tipo	Estado	Notas
Unit tests	⚠ Básicos	Estructura lista, faltan más casos
Integration tests	⚠ Flujo completo	`test_complete_flow.py` básico
E2E tests	⏳ No implementado	Requiere Selenium/Playwright

Tests Disponibles

```
# Ejecutar todos los tests
pytest

# Ejecutar con cobertura
pytest --cov=backend

# Tests específicos
pytest backend/tests/integration/test_complete_flow.py
pytest backend/tests/unit/agents/
```

Funcionalidades Implementadas

Core (100%)

- ✅ Sistema multi-agente con orquestador
- ✅ Autenticación JWT completa
- ✅ CRUD de productos (Service + GraphQL)
- ✅ CRUD de órdenes con transacciones atómicas
- ✅ Manejo de stock (descontar/restaurar)
- ✅ Chat contextual con Alex
- ✅ Adaptación de estilo de usuario
- ✅ RAG con ChromaDB
- ✅ Rate limiting
- ✅ Sesiones con Redis

Avanzadas (80%)

- ✅ Reconocimiento de intenciones con LLM
- ✅ Detección de estilo de comunicación
- ✅ Slot filling (memoria de conversación)
- ✅ Transferencias entre agentes
- ✅ Manejo de objeciones de precio
- ✅ Anti-alucinación (solo productos reales)
- ⚠️ Reconocimiento visual (cliente listo, servicio externo)

Pendientes (20%)

- ⏳ Agente 1 - Registro de inventario desde fotos
- ⏳ WebSockets para chat en tiempo real
- ⏳ Notificaciones push
- ⏳ Dashboard de administración
- ⏳ Análisis de métricas de ventas

Próximos Pasos Recomendados

Prioridad Alta

1. **Integrar Agente 1**: Crear cliente HTTP para servicio de registro de productos
2. **Mejorar Tests**: Aumentar cobertura de tests unitarios e integración
3. **Documentación API**: Generar docs automáticas con GraphQL introspection

Prioridad Media

4. **WebSockets**: Implementar suscripciones GraphQL para chat en tiempo real
5. **Cache inteligente**: Cachear resultados de búsqueda frecuentes
6. **Métricas**: Implementar tracking de conversiones del agente

Prioridad Baja

7. **i18n**: Soporte multi-idioma (español, inglés)
8. **A/B testing**: Probar diferentes prompts de persuasión
9. **ML interno**: Entrenar clasificador de intenciones propio

Métricas del Sistema

Métrica	Valor Actual	Objetivo
Latencia query (avg)	~500-2000ms	<1000ms
Disponibilidad	99% (local)	99.9%
Cobertura tests	~40%	>80%
Agente transfer accuracy	~85%	>90%
Conversión simulada	No medido	Implementar

Documento Generado: Febrero 2026
Responsable: Grupo 3 - Backend Agéntico
Estado: Sistema Operativo y Listo para Demo