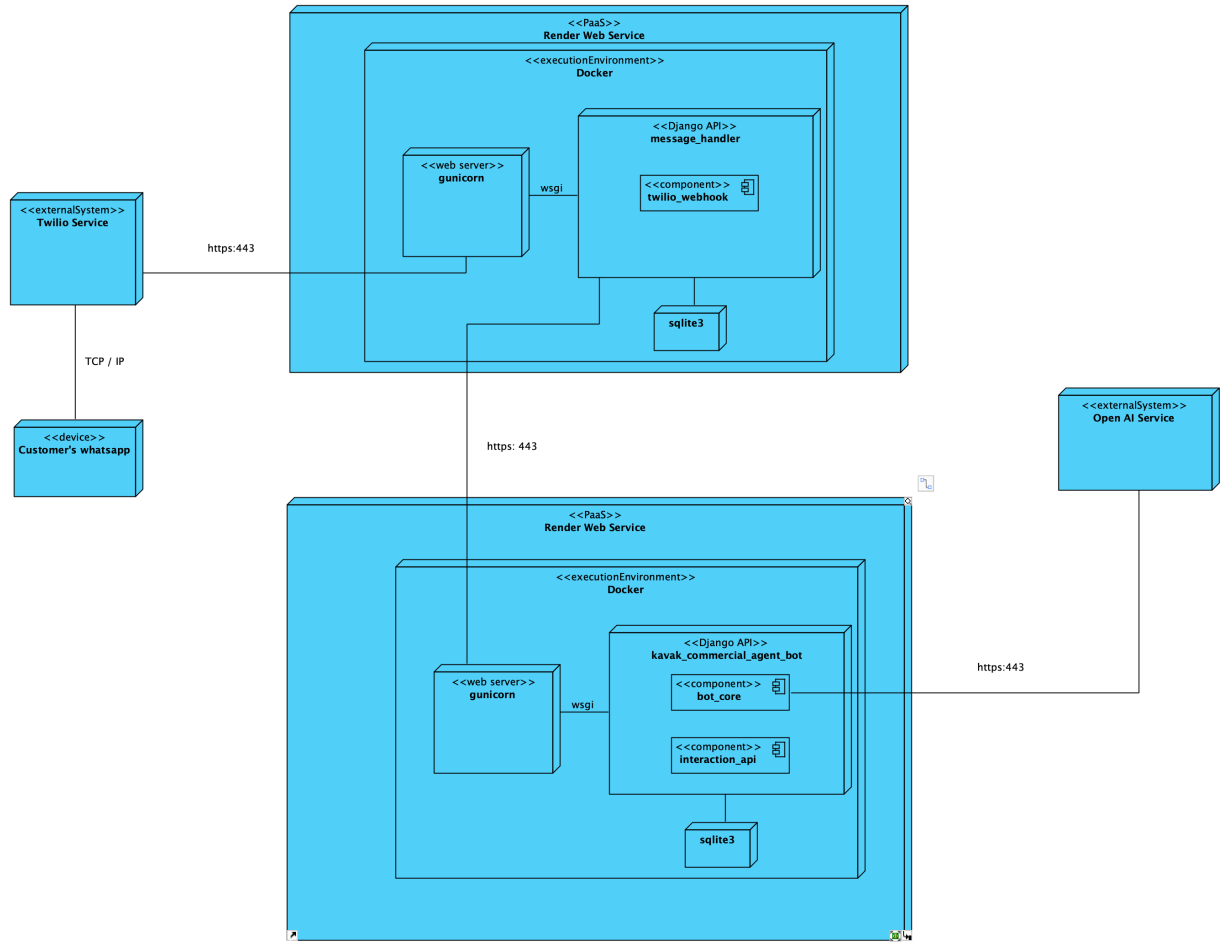
# Agente comercial AI Kavak

## Diagrama de alto nivel sobre componentes y arquitectura




Este diagrama de despliegue ilustra la arquitectura de alto nivel para el sistema del Agente Comercial de IA de Kavak. Muestra los principales componentes de software, cómo están distribuidos en sus entornos de ejecución y cómo interactúan entre sí y con servicios externos.

### Visión general de la arquitectura

El sistema está diseñado con una arquitectura desacoplada basada en dos servicios principales, cada uno con responsabilidades específicas, y se apoya en servicios externos clave para la comunicación y la inteligencia artificial (LLM models).

### Componentes detallados

A continuación, se describen los componentes clave visualizados en el diagrama:

### 1. Cliente (usuario de whatsApp)

- **Nodo:** `<<device>> Customer's whatsapp`
- **Descripción:** Representa al usuario final que interactúa con el bot a través de la aplicación WhatsApp en su dispositivo. Es el punto de inicio y fin de la interacción.

### 2. Servicio Twilio

- **Nodo:** `<<externalSystem>> Twilio Service`
- **Descripción:** Actúa como la pasarela entre la plataforma de WhatsApp y nuestra aplicación. Gestiona la recepción de mensajes del cliente vía WhatsApp y la entrega de las respuestas del bot de vuelta al cliente.

### 3. Servicio `kavak_message_handler` (API receptora de mensajes)

- **Nodo Contenedor Principal:** `<<PaaS>> Render Web Service`
- **Descripción:** Este es el primer punto de contacto de la infraestructura. Su principal responsabilidad es recibir las solicitudes webhook desde el servicio de Twilio, registrar la interacción y luego reenviar el mensaje del usuario al servicio de lógica del bot (`commercial_agent_bot`) para su procesamiento. También es responsable de recibir la respuesta final del bot y enviarla de vuelta a Twilio para entregarla al cliente.
- **Entorno de ejecución:**
  - Desplegado en **Render** (Plataforma como Servicio).
  - Dentro de un `<<executionEnvironment>> Docker` para encapsular la aplicación y sus dependencias.
  - Utiliza `<<web server>> gunicorn` como servidor WSGI HTTP para la aplicación Django.
- **Componente principal de la aplicación Django (`<<Django API>> message_handler`):**
  - `<<component>> twilio_webhook`: Aplicación Django interna que maneja específicamente los webhooks entrantes de Twilio.
- **Base de datos:**
  - `sqlite3`: Utiliza una base de datos SQLite para almacenamiento local de la comunicación recibida

### 4. Servicio `commercial_agent_bot` (API de Lógica del Bot)

- **Nodo contenedor principal:** `<<PaaS>> Render Web Service`
- **Descripción:** Este servicio alberga el "cerebro" del agente comercial. Recibe el mensaje del usuario (reenviado por `kavak_message_handler` vía HTTPS en el puerto 443), orquesta el proceso de entendimiento de la intención del usuario, invoca a los agentes especializados para cada tarea, interactúa con el servicio de OpenAI para la generación de lenguaje natural y la comprensión, y formula la respuesta final.
- **Entorno de ejecución:**
  - Desplegado en **Render**.
  - Dentro de un `<<executionEnvironment>> Docker`.
  - Utiliza `<<web server>> gunicorn` como servidor WSGI HTTP.
- **Componentes principales de la aplicación Django (`<<Django API>> kavak_commercial_agent_bot`):**
  - `<<component>> interaction_api`: Aplicación Django que expone el endpoint para recibir las solicitudes del servicio `kavak_message_handler`.
  - `<<component>> bot_core`: Módulo central que contiene la lógica de orquestación, los diferentes agentes (ej. `CarRecommendationAgent`, `FinancingAgent`, etc.), y la lógica para construir prompts y procesar respuestas de la IA.
- **Base de Datos:**
  - `sqlite3`: Utiliza una base de datos SQLite para su propio almacenamiento, que podría incluir logs de conversación detallados, plantillas de prompts.

### 5. Servicio Open AI

- **Nodo:** `<<externalSystem>> Open AI Service`
- **Descripción:** Es el servicio de Large Language Model (LLM) externo. El componente `bot_core` del servicio `commercial_agent_bot` realiza llamadas API (vía HTTPS en el puerto 443) a este servicio para:
  1. Interpretar la intención del mensaje del usuario.
  2. Generar respuestas coherentes y contextualizadas para el usuario.

## Secuencia de comunicación principal

1. El **Cliente** envía un mensaje por WhatsApp.
2. El **Servicio Twilio** recibe el mensaje y lo reenvía como un webhook HTTPS al servicio `kavak_message_handler`.
3. `kavak_message_handler` procesa la solicitud y reenvía el mensaje relevante al servicio `commercial_agent_bot` mediante una llamada API HTTPS.
4. `commercial_agent_bot` (específicamente el `bot_core`) interactúa con el **Servicio Open AI** (vía HTTPS) para entender la intención y/o generar una respuesta.
5. `commercial_agent_bot` devuelve la respuesta formulada a `kavak_message_handler`.
6. `kavak_message_handler` envía la respuesta al **Servicio Twilio**, el cual la entrega al **Cliente** en WhatsApp.

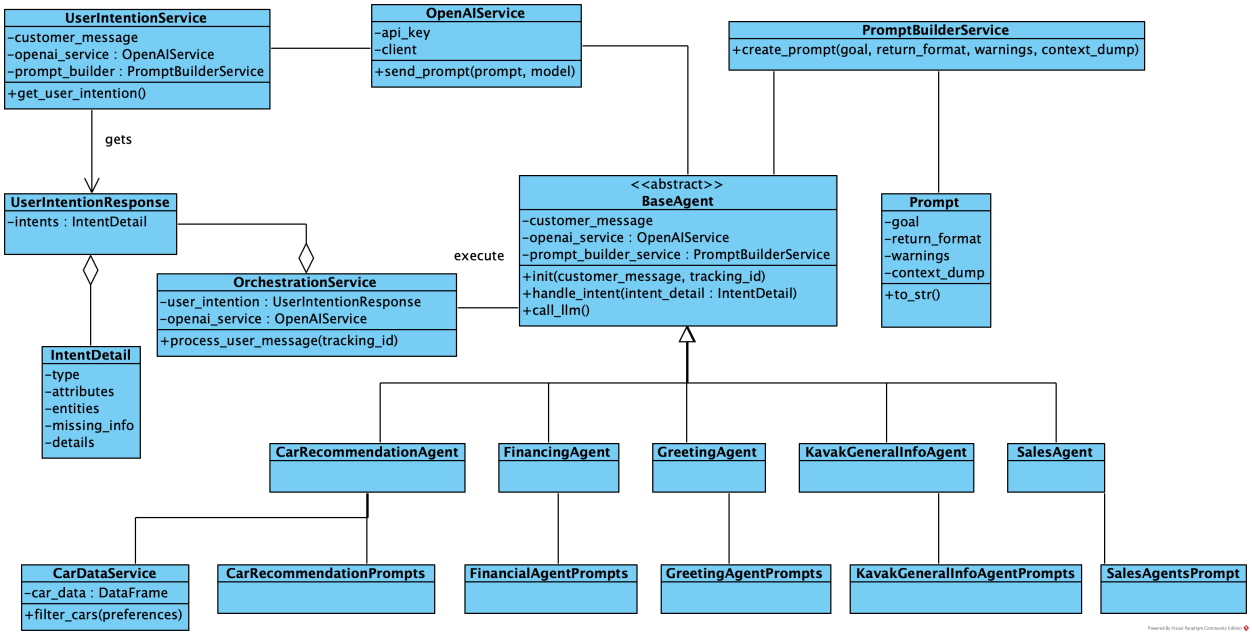
## Tecnologías de despliegue

- Ambos servicios (`kavak_message_handler` y `commercial_agent_bot`) están desplegados en la plataforma **Render**, utilizando **Docker** para la contenerización y **Gunicorn** como servidor WSGI para las aplicaciones **Django**.
- La comunicación interna entre Gunicorn y las aplicaciones Django se realiza a través de la interfaz **WSGI**.

El diagrama muestra una arquitectura modular y distribuida, donde cada servicio tiene responsabilidades claras. La separación entre la recepción de mensajes y la lógica del bot permite un desarrollo y escalado más manejable de las diferentes partes del sistema. Las dependencias externas están claramente identificadas, facilitando la comprensión del ecosistema completo de la aplicación.

# Diagrama sobre agentes

Este diagrama representa el “cerebro” del bot. Explicaré a continuación lo más relevante.




## Componentes Principales

Las clases clave y sus responsabilidades son las siguientes:

- `UserIntentionService`:
  - Servicio dedicado a **interpretar la intención** del usuario.
  - Utiliza `PromptBuilderService` y las constantes de `UserIntentionPrompts` para construir un prompt específico para el reconocimiento de intenciones.
  - Llama a `OpenAIService` para ejecutar este prompt en el LLM (OpenAI).
  - **Valida y estructura** la respuesta JSON del LLM usando el modelo Pydantic `UserIntentionResponse`.
- `OrchestrationService`:
  - Actúa como el **coordinador central** del `bot_core`.
  - Su método principal, `process_user_message(tracking_id)`, recibe el mensaje original del cliente y un ID de seguimiento.
  - Utiliza el `UserIntentionService` para realizar la **primera llamada al LLM (OpenAI)** y obtener la intención del usuario de forma estructurada.
  - Analiza la respuesta (`UserIntentionResponse`) para identificar una o **múltiples intenciones**(`IntentDetail`).
  - Para cada intención detectada, **instancia y ejecuta** (`execute`) el Agente especializado correspondiente (que hereda de `BaseAgent`).
  - Finalmente, **sintetiza** las respuestas de todos los agentes ejecutados para formar la respuesta final que se enviará al usuario.
- `UserIntentionResponse` / `IntentDetail`:
  - Modelos Pydantic que definen la **estructura de datos** para la respuesta del LLM de reconocimiento de intención. Garantizan que la información sobre las intenciones, atributos, entidades, etc., sea consistente y fácil de manejar.
- `BaseAgent` (`<<abstract>>`):
  - Clase base **abstracta** para todos los agentes especializados. Define la interfaz común (`handle_intent`, `call_llm`) y contiene atributos compartidos como el `tracking_id`, el mensaje original del cliente y referencias a servicios comunes

(`OpenAIService`, `PromptBuilderService`).
    - Centraliza la lógica para llamar al LLM (`call_llm`) a través del `OpenAIService`.
- **Agentes Concretos** (`CarRecommendationAgent`, `FinancingAgent`, `GreetingAgent`, `KavakGeneralInfoAgent`, `SalesAgent`):
    - Heredan de `BaseAgent` e implementan la lógica específica para manejar una **intención particular**.
    - El método `handle_intent(intent_detail, tracking_id)` recibe los detalles de la intención relevante.
    - Utilizan **"Tools"** (otros servicios) como `CarDataService` (para recomendaciones) o acceden a contexto específico (como el HTML en `KavakGeneralInfoAgent`).
    - Usan `PromptBuilderService` y sus clases de `Prompts` asociadas para construir un **prompt detallado y contextualizado** para la generación de la respuesta final.
    - Invocan la **segunda llamada al LLM (OpenAI)** a través del método heredado `call_llm` para obtener el texto de la respuesta.
    - Devuelven la respuesta generada al `OrchestrationService`.
- **Servicios:**
    - `OpenAIService`: Fachada que encapsula la interacción directa con la API de OpenAI (método `send_prompt`). Usado tanto por `UserIntentionService` como por los Agentes.
    - `PromptBuilderService`: Ayuda a construir objetos `Prompt` estructurados, separando el objetivo, formato, contexto y advertencias, antes de generar el string final del prompt (`Prompt.to_str()`).
    - `CarDataService`: Servicio específico que actúa como una herramienta para el `CarRecommendationAgent`, manejando la carga y filtrado de datos del catálogo de autos (desde un CSV en este caso).
    - **Clases** `Prompts` (`CarRecommendationPrompts`, `FinancingPrompts`, etc.): Actúan como contenedores de configuración para las plantillas y secciones de texto de los prompts, promoviendo la organización y facilitando futuras modificaciones.

## Secuencia de ejecución

1. Llama a `UserIntentionService` para obtener el objeto `UserIntentionResponse`.
2. `OrchestrationService` recibe el `UserIntentionResponse`
3. Itera sobre cada `IntentDetail` en `UserIntentionResponse.intents`.
4. Para un `IntentDetail` (ej. tipo `car_recommendation`), instancia `CarRecommendationAgent`.
5. Llama a `CarRecommendationAgent.handle_intent()`.
6. El agente usa `CarDataService` para filtrar autos y `PromptBuilderService` (con `CarRecommendationPrompts`) para crear el prompt de respuesta.
7. El agente llama a `OpenAIService` (vía `call_llm`) con el prompt construido.
8. El agente devuelve el texto de respuesta al `OrchestrationService`.
9. El `OrchestrationService` combina esta respuesta (y las de otros agentes si hubo múltiples intenciones) y la retorna.

## Manejo de Prompts y Seguimiento

- Los **Prompts** se gestionan de forma organizada usando el `PromptBuilderService`, la clase `Prompt` y las clases de constantes específicas (`Prompts`), permitiendo una construcción estructurada y mantenible.
- El `tracking_id` se propaga desde el inicio hasta los agentes y potencialmente a los servicios que estos usan, lo que es fundamental para el logging y la trazabilidad de cada solicitud a través del `bot_core`.

El diagrama de clases demuestra una arquitectura orientada a objetos clara y modular para el `bot_core`. La separación en servicios (Orquestador, Intención, OpenAI, Datos, Prompts) y el uso de un patrón de Agentes basado en herencia (`BaseAgent`)

permiten una buena separación de responsabilidades, facilitan la extensibilidad (añadir nuevos agentes/intenciones). Esta estructura soporta eficazmente el flujo de doble LLM (Intención → Respuesta) y el uso de herramientas contextuales.

# Propuesta de Roadmap y Backlog: Agente Comercial Kavak a Producción

Esta propuesta describe el camino para llevar el prototipo actual del Agente Comercial de IA de Kavak a un entorno de producción robusto, escalable y mantenible. También aborda cómo se evaluará el desempeño del agente y cómo se asegurará la calidad en futuras versiones.

## Fase 0: MVP Actual (Entorno de Desarrollo/Prueba)

- **Estado:** Prototipo funcional en un PaaS (Render) usando Docker, Gunicorn y SQLite3 para dos servicios Django (`kavak_message_handler` y `commercial_agent_bot`). Lógica básica de agentes implementada.
- **Objetivo:** Demostrar la viabilidad del concepto y la funcionalidad central.

## Fase 1: Preparación para Producción (V1.0)

- **Objetivo:** Migrar a una infraestructura de nube más robusta y configurable (AWS), implementar bases de datos de producción, establecer CI/CD básico y mecanismos iniciales de monitoreo y evaluación.

1. **Mejoras en la Inteligencia del Agente:**

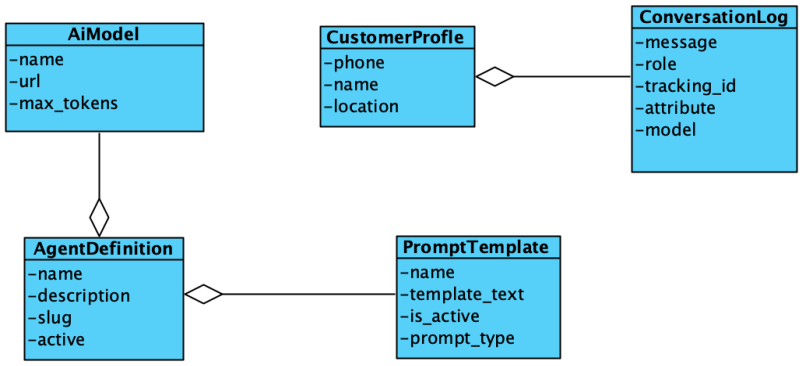



Diagrama de clases con los modelos que permitirían mejorar la inteligencia global del bot

   - **Cambio de modelo dependiendo del agente:**
     - Dependiendo del agente, esto permitirá usar los agentes que tienen mejor performance o que son mejores para ejecutar cálculos. Ayudando a mejorar cada agente por separado
   - **Gestión dinámica de prompts**
     - Desde la base de datos de `bot_core` (como se propuso).
     - Teniendo una relación de los Agentes con los Prompts se podría modificar prompts sin tener que hacer deploy, así se podría de forma más fácil pruebas para nuevos prompts.
   - **Historial de mensajes:**
     - En el MVP el envío de mensajes es stateless, el LLM no tiene contexto de las respuestas anteriores, persistiendo los logs de la conversación así como el rol nos va a permitir inyectar cada petición con el historial necesario.
2. **Diseño y Configuración de Infraestructura en AWS:**
   - **Servicios de Cómputo:**

    - Migrar los servicios Django (`kavak_message_handler`, `commercial_agent_bot`) a **AWS ECS (Elastic Container Service)** con Fargate para despliegues sin servidor de contenedores Docker.
    - Utilizar **Amazon ECR (Elastic Container Registry)** para almacenar las imágenes Docker.
  - **Red y Enrutamiento:**
    - Configurar **Amazon API Gateway** como el punto de entrada para las solicitudes de Twilio y para la comunicación entre los dos servicios. Gestionará autenticación, enrutamiento y puede ayudar con el versionado y throttling inicial.
  - **Bases de Datos:**
    - `kavak_message_handler`**:** Migrar a **Amazon DynamoDB**. Apropiado para el alto volumen de escritura de mensajes entrantes y logs, con flexibilidad de esquema. Se almacenará: `tracking_id` (clave primaria), `customer_phone`, `received_message_timestamp`, `raw_user_message`, `status` (`received`, `processed`, `replied`, `failed`), `final_reply` aplica).
    - `commercial_agent_bot`**:** Migrar a **Amazon RDS para PostgreSQL** Apropiado para datos relacionales como `PromptsTemplate`, `AgentDefinitions`, `AIModels`, `ConversationLog` (reemplazando y expandiendo el actual `ConversationLog`), `CustomerProfiles`.
  - **Gestión de Colas:**
    - Implementar **Amazon SQS (Simple Queue Service)** entre `kavak_message_handler` y `commercial_agent_bot`. `kavak_message_handler` publicará los mensajes entrantes en una cola SQS. `commercial_agent_bot` tendrá workers consumiendo de esta cola a una tasa controlada para respetar los límites de la API de OpenAI y manejar picos de carga.
3. **Refactorización y Mejoras de la Aplicación:**
  - Adaptar las aplicaciones Django para usar las nuevas bases de datos (DynamoDB y PostgreSQL).
  - Implementar la lógica de publicación/consumo de mensajes con SQS.
  - Manejo de configuración y secretos usando **AWS Secrets Manager** o variables de entorno en ECS.
4. **Implementación de CI/CD Básico:**
  - Configurar un pipeline en **GitHub Actions** (o GitLab CI/AWS CodePipeline).
  - Pasos: Linting, ejecución de pruebas unitarias, construcción de imágenes Docker, push a ECR, despliegue en ECS.
5. **Pruebas Fundamentales:**
  - Desarrollar **pruebas unitarias** para los servicios y lógica crítica no dependiente de LLM.
  - Crear un **conjunto de datos de prueba** inicial con mensajes de ejemplo y respuestas esperadas clave. Esto nos ayudará a evaluar el desempeño de diferentes modelos / refinación de prompts / mejora en los agentes.
6. **Monitoreo y Logging Básico:**
  - Utilizar **Amazon CloudWatch** para logs de aplicación (desde Gunicorn/Django), métricas de ECS, SQS, API Gateway y bases de datos.
  - Crear alertas básicas para errores críticos y utilización de recursos.

## Fase 2: Optimización y Evaluación (V1.1)

- **Objetivo:** Optimizar el rendimiento, costos, implementar un sistema de evaluación del agente más robusto y mejorar la observabilidad.
  1. **Optimización de Tokens y Costos de LLM:**
    - Analizar el uso de tokens por tipo de agente/intención.
    - Refinar prompts para reducir la verbosidad sin perder calidad.

- Implementar caché para respuestas a preguntas frecuentes.

2. **Sistema de Evaluación del Desempeño del Agente:**
   - **Métricas Cuantitativas:** Implementar tracking de:
     - Tiempo de respuesta (end-to-end, procesamiento LLM).
     - Tasa de errores (fallos de API, intenciones no manejadas, errores de SQS).
     - Consumo de tokens promedio por interacción.
     - Volumen de mensajes procesados / en cola.
   - **Métricas Cualitativas (Inicial):**
     - Revisión manual periódica de una muestra de conversaciones del `ConversationLog` en `bot_core`.
     - Implementar un mecanismo simple de feedback del usuario en WhatsApp ("¿Te fue útil? 👍/👎").
   - **Dashboard de Monitoreo:** Crear un dashboard básico en CloudWatch o Grafana con las métricas clave.
3. **Pruebas de Regresión Automatizadas:**
   - Automatizar la ejecución de pruebas contra el set de mensajes de prueba como parte del CI/CD.
   - Implementar comparaciones básicas de respuestas (ej. presencia de keywords, longitud esperada).
4. **Seguridad y Cumplimiento:**
   - Revisar configuraciones de seguridad en AWS (IAM roles, security groups, etc.).
   - Asegurar el manejo adecuado de datos sensibles del cliente.

### Fase 3: Características Avanzadas y Escalado (V2.0)

- **Objetivo:** Mejorar la inteligencia del bot, la experiencia del usuario y la eficiencia operativa.

1. Mejorar inteligencia del agente
   - Integración con otros sistemas de Kavak (CRM, inventario en tiempo real, etc).
2. **Evaluación Avanzada del Desempeño:**
   - Implementar "LLM-as-a-Judge" para evaluación cualitativa automatizada. De esta forma usando otro modelo como *jurado* podremos obtener métricas de desempeño de nuestro bot
   - Análisis de sentimiento de las respuestas del usuario.
3. **Escalado y Resiliencia Avanzados:**
   - Auto-scaling configurado para ECS y workers SQS.
   - Explorar EKS si la complejidad de la orquestación lo amerita.
4. **Herramientas de Soporte y Operaciones:**
   - Interfaz administrativa para revisar conversaciones, gestionar prompts (si son dinámicos), y monitorear el estado del bot.

---

## Backlog Inicial (Priorizado para Fase 1 y 2)

Este backlog representa las tareas clave. Se pueden desglosar en historias de usuario más pequeñas.

### Infraestructura y Despliegue (Fase 1)

- **ÉPICA: Mejorar Modelos de Datos y Lógica de `bot_core`**
  - Expandir el modelo `ConversationLog` para trazabilidad completa (incluyendo prompts enviados, respuestas de LLM, tokens usados)
  - Uso de `ConversationLog` para historial de conversaciones para contexto del agente

    - Diseñar e implementar `CustomerProfile` para almacenar información del cliente y enriquecer contexto.
    - Implementar modelos `PromptTemplate`, `AgentDefinition`, `AIModel` para gestión dinámica.
- **ÉPICA: Configurar Infraestructura Base en AWS**
    - Definir VPC, subredes, security groups.
    - Configurar roles IAM para los servicios.
    - Crear clúster de ECS y definiciones de tareas para `kavak_message_handler` y `commercial_agent_bot`.
    - Configurar ECR y subir imágenes Docker iniciales.
    - Configurar API Gateway con rutas a los servicios de ECS.
    - Configurar SQS para la cola de mensajes entre servicios.
    - Configurar AWS Secrets Manager para claves de API y contraseñas de BD.
    - Configurar CloudWatch Logs para todos los servicios.
- **ÉPICA: Configurar Bases de Datos de Producción**
    - Aprovisionar instancia de Amazon RDS PostgreSQL para `commercial_agent_bot`.
    - Crear tabla en Amazon DynamoDB para `kavak_message_handler`.
    - Migrar esquemas de BD de SQLite a PostgreSQL y diseñar tabla DynamoDB.
    - Configurar conexiones seguras desde ECS a las bases de datos.

## Aplicación y Lógica del Bot (Fase 1 y 2)

- **ÉPICA: Adaptar Aplicaciones para Producción en AWS**
    - Modificar `kavak_message_handler` para publicar en SQS y escribir en DynamoDB.
    - Modificar `commercial_agent_bot` para consumir de SQS y usar PostgreSQL.
    - Implementar lógica de reintentos y manejo de errores para SQS y llamadas a OpenAI.
    - Asegurar que la configuración se carga desde variables de entorno / Secrets Manager.
    - Implementar workers SQS en `commercial_agent_bot` con control de concurrencia para respetar rate limits de OpenAI.
- **ÉPICA: Mejorar Modelos de Datos y Lógica de `bot_core`**
    - Expandir el modelo `ConversationLog` en PostgreSQL para trazabilidad completa (incluyendo prompts enviados, respuestas de LLM, tokens usados, costos estimados).
    - Diseñar e implementar `CustomerProfile` para almacenar información del cliente y enriquecer contexto.
    - (Fase 2/V2.0) Implementar modelos `PromptTemplate`, `AgentDefinition`, `AIModel` en PostgreSQL para gestión dinámica.
- **TAREA: Optimización de Tokens (Fase 2)**
    - Revisar todos los prompts actuales para eficiencia de tokens.
    - Implementar logging detallado del uso de tokens por cada llamada a OpenAI.

## Pruebas y Calidad (Fase 1 y 2)

- **ÉPICA: Establecer Fundamentos de Pruebas**
    - Desarrollar pruebas unitarias para lógica de negocio crítica en ambos servicios (mínimo 60% de cobertura).
    - Desarrollar pruebas de integración para la comunicación entre servicios (API Gateway → ECS, SQS).
    - Crear y mantener un "golden set" de al menos 50 interacciones de prueba cubriendo los principales casos de uso.
- **TAREA: Automatizar Pruebas de Regresión (Fase 2)**
    - Integrar la ejecución del "golden set" en el pipeline de CI/CD.

- Definir criterios de éxito/fallo para las pruebas del "golden set" (inicialmente puede ser revisión manual del output, luego automatizar comparaciones).

### CI/CD (Fase 1)

- **ÉPICA: Implementar Pipeline Básico de CI/CD**
  - Configurar repositorio Git (GitHub/GitLab).
  - Crear pipeline en GitHub Actions (o similar) que incluya:
    - Checkout de código.
    - Linting (ej. Flake8, Black).
    - Ejecución de pruebas unitarias.
    - Construcción de imágenes Docker.
    - Push de imágenes a ECR.
    - Despliegue en ECS (actualización de servicios).

### Monitoreo, Evaluación y Operaciones (Fase 2)

- **ÉPICA: Implementar Sistema de Evaluación y Monitoreo V1**
  - Configurar dashboards en CloudWatch para métricas clave (errores, latencia, uso de SQS, uso de CPU/Memoria en ECS, tokens OpenAI).
  - Implementar logging estructurado para facilitar búsquedas y análisis.
  - Crear mecanismo para recolección de feedback del usuario (ej. 👍/👎 en WhatsApp).
  - Definir proceso para revisión manual periódica de conversaciones.
- **TAREA: Documentación Operativa (Fase 1/2)**
  - Crear runbooks para despliegues, rollback, y manejo de incidentes comunes.