K.A.B.O.T. – Knowledge-augmented Associative Brain with Orchestrated Temporal-awareness Thinking

- **K**nowledge-augmented → porque no es solo un LLM: tiene memoria enriquecida, datos persistentes y contexto.
- **A**ssociative Memory → puede hacer asociaciones de hechos y contexto en tiempo real.
- **B**rain-Inspired → su arquitectura jerárquica y modular se inspira en el funcionamiento del cerebro humano.
- **O**rchestrated Reasoning → un orquestador central dirige el flujo de intenciones, decisiones y actualizaciones.
- **T**emporal-awareness + **T**hinking → no solo recuerda, sino que piensa con conciencia del tiempo, actualiza recuerdos y razona en función del contexto histórico.

K.A.B.O.T. no es solo un bot. Es un sistema cognitivo distribuido con memoria jerárquica, razonamiento progresivo y actualización semántica en tiempo real.

KABOT - Arquitectura y razonamiento técnico

Este reto técnico despertó particularmente mi interés gracias a la libertad que ofrecía para experimentar. Por ello, decidí llevar a la práctica algunas de mis ideas más disruptivas y construir una arquitectura inspirada en el funcionamiento del cerebro humano.

El cerebro utiliza múltiples formas de memoria, entre ellas la memoria de trabajo (o memoria a corto plazo) y la memoria a largo plazo. Esta distinción ha servido de base para la creación de sistemas computacionales de almacenamiento como la memoria RAM y el disco duro. Durante el sueño, el cerebro humano ejecuta un proceso de consolidación en el que filtra las experiencias del día, determina cuáles son relevantes para almacenar a largo plazo y cuáles pueden descartarse. Además, a diferencia de un sistema digital, el cerebro no guarda recuerdos exactos sino que extrae los detalles más significativos y luego los reconstruye bajo demanda. Esta capacidad de abstracción y reconstrucción fue una fuente de inspiración directa para el sistema que propuse.

Con esta base conceptual, diseñé una arquitectura que separa claramente distintos tipos de memoria, cada uno con un propósito específico:

- **Memoria de trabajo (WorkingMemory)**: Utiliza una caché para almacenar las interacciones más recientes con el usuario. Esto permite consultas rápidas y una conversación más coherente, dado que el LLM puede acceder a contexto inmediato.

- **Memoria de resumen (SummaryMemory)**: Cuando una conversación finaliza, la memoria de trabajo es condensada y almacenada como un solo párrafo en una base de datos no relacional. Esta representación resumida permite al sistema recordar eventos pasados sin necesidad de procesar todo el historial nuevamente.

- **Memoria de hechos (FactsMemory)**: También derivada del contenido reciente, pero con un enfoque en almacenar información estructurada como preferencias, relaciones, intereses o datos personales. Se guarda en formato JSON, haciéndola fácilmente consultable y útil para tareas más específicas.

- **Memoria episódica (EpisodicMemory)**: A diferencia de las anteriores, esta almacena la conversación completa sin procesar. Es útil cuando se necesita recuperar el historial íntegro de una sesión para resolver ambigüedades o responder con precisión ante referencias pasadas complejas.

Surge entonces una pregunta válida: ¿por qué dividir la memoria en tantas capas? ¿No bastaría con almacenar todo el historial? La respuesta es eficiencia. Eficiencia en costos computacionales, velocidad de respuesta y consumo de tokens en las llamadas al LLM.

Cuando un mensaje llega al webhook, el **CognitiveOrchestrator** se encarga de gestionar el flujo. Como primer paso, solicita datos de SummaryMemory y FactsMemory, y los coloca en WorkingMemory para uso inmediato. A lo largo de la conversación, la memoria de trabajo se enriquece. Al cierre de la sesión, el ExitHandler transfiere lo relevante a las memorias de largo plazo y limpia la caché, dejando todo listo para futuras conversaciones.

Además de la gestión de memoria, el CognitiveOrchestrator se apoya en diversos **handlers** para procesar intenciones específicas, las cuales son determinadas por el IntentionHandler, que analiza el mensaje del usuario y define cuál flujo seguir. Las intenciones cubren:

- **SearchHandler**: Transforma el mensaje en embeddings para realizar búsquedas vectoriales en OpenSearch y, en paralelo, ejecuta búsquedas por términos. Posteriormente, los resultados se resumen y presentan al usuario.

- **InfoHandler**: Provee información institucional (como datos sobre Kavak) en lenguaje natural.

- **FinanceHandler**: Responde a preguntas sobre financiamiento, haciendo cálculos según los datos proporcionados.

- **ExitHandler**: Se encarga de persistir la conversación y limpiar el contexto al final de la sesión.

En resumen, esta arquitectura está optimizada para ofrecer respuestas naturales y relevantes, sin comprometer la eficiencia. El diseño permite manejar la conversación como una entidad cognitiva que recuerda, abstrae, deduce y conversa de manera contextual, replicando —en cierta medida— cómo lo hace un ser humano.

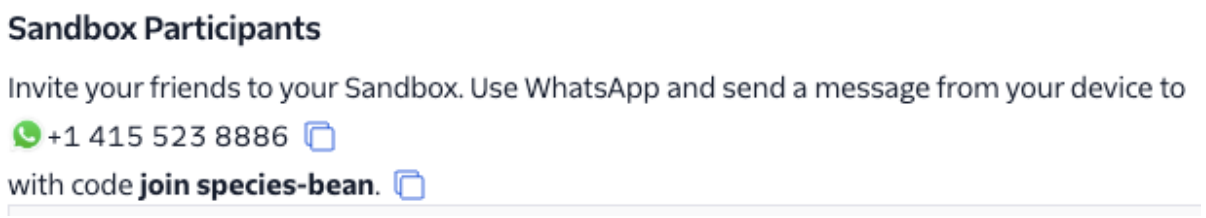
¿El resultado? Un agente conversacional más empático, efectivo y escalable. Un K.A.B.O.T.

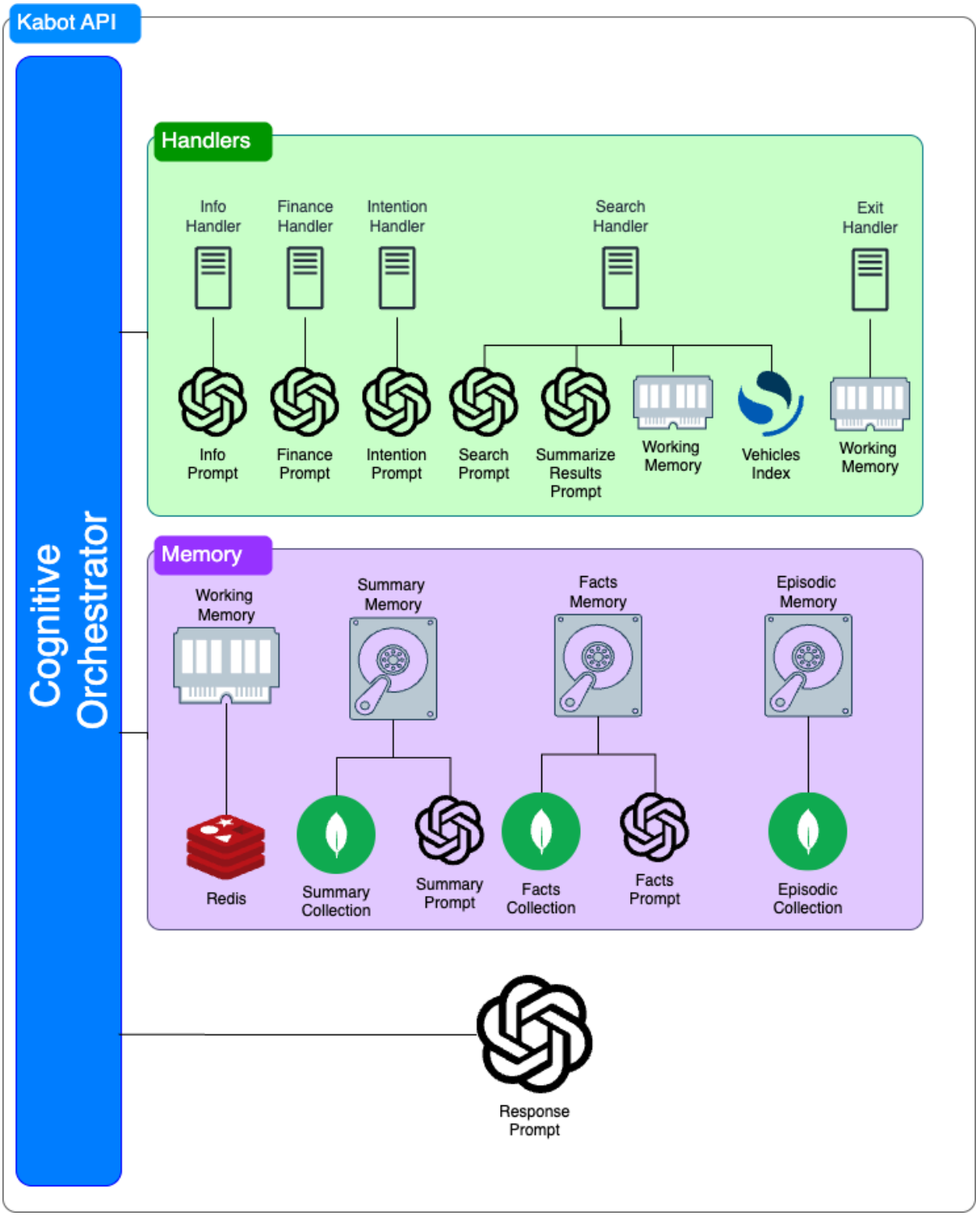
Ejemplos de memorias:

```
kabot> db.fact_memory.find({ "whatsapp_id": "5215578771322" }).pretty()
[
  {
    _id: ObjectId('682b9a365e5f71db67fb4328'),
    whatsapp_id: '5215578771322',
    facts: {
      name: 'Leonardo',
      age: 32,
      location: 'México',
      pets: { cats: [ 'Chata', 'Chimino', 'Canelita' ] },
      interests: [ 'autos', 'Kavak', 'Mazda 3', 'Ford', 'BMW', 'SUV' ],
      auto_preferences: {
        brands: [ 'Mazda', 'Ford', 'BMW' ],
        specific_models: [ 'Mazda 3', 'BMW Serie 3 2018' ]
      },
      recent_auto_inquiry: {
        make: 'BMW',
        model: 'Serie 3',
        year: 2018,
        price: 421999,
        km: 63290,
        features: { bluetooth: true, car_play: false }
      },
      financing_interest: { engagement: 110000, estimated_monthly_payment: 12373 },
      recent_vehicle_inquiries: [
        {
          make: 'Ford',
          model: 'Escape',
          year: 2019,
          price: 280999,
          km: 71677,
          features: { bluetooth: true, car_play: true }
        },
        {
          make: 'Ford',
          model: 'EcoSport',
          year: 2020,
          price: 285999,
          km: 27000,
          features: { bluetooth: true, car_play: true }
        },
        {
          make: 'Ford',
          model: 'Figo',
          year: 2020,
          price: 188999,
          km: 26011,
          features: { bluetooth: true, car_play: false }
        }
      ]
    },
    last_updated: '2025-05-19T21:18:05.546205'
  }
]
kabot> db.summary_memory.find({ "whatsapp_id": "5215578771322" }).pretty()
```

```
kabot> db.summary_memory.find({ "whatsapp_id": "5215578771322" }).pretty()
[
  {
    _id: ObjectId('682b9a325e5f71db67fb4325'),
    whatsapp_id: '5215578771322',
    last_updated: '2025-05-20T00:40:41.370104',
    summary: 'Leonardo, de 32 años y residente en México, tiene un interés en autos, especialmente en marcas como Mazda
, Ford, BMW y Dodge. Ha mostrado interés en un BMW Serie 3 2018 y ha explorado opciones de financiamiento. En su recien
te interacción, el asistente le ofreció ayuda con temas relacionados con autos, pero la conversación fue breve y amigab
le. Leonardo tiene tres gatos: Chata, Chimino y Canelita.'
  }
]
```

Para pruebas:




Kabot API
Cognitive Orchestrator
Handlers
Info Handler
Finance Handler
Intention Handler
Search Handler
Exit Handler
Info Prompt
Finance Prompt
Intention Prompt
Search Prompt
Summarize Results Prompt
Working Memory
Vehicles Index
Working Memory
Memory
Working Memory
Summary Memory
Facts Memory
Episodic Memory
Redis
Summary Collection
Summary Prompt
Facts Collection
Facts Prompt
Episodic Collection
Response Prompt

Roadmap

Fase 1: Modularización y definición de contratos

Objetivo: Garantizar independencia entre servicios.

- Separar cada memoria (working, episodic, summary, fact) en su propio microservicio con su propia API.
- Exponer los handlers (intention, finance, info, exit) como un solo servicio orquestado con endpoints dedicados.
- Dejar search_handler como un microservicio independiente que expone /search y consume OpenSearch + LLM.
- Documentar todos los contratos en OpenAPI/Swagger.
- Definir un esquema estándar de payloads y errores para comunicación interna.

Fase 2: Infraestructura de comunicación y resiliencia

Objetivo: Asegurar comunicación eficiente entre microservicios.

- Establecer un API Gateway o Service Mesh (ej. Istio o Linkerd).
- Incluir timeouts, retries y circuit breakers entre servicios.
- Configurar Redis como un Event Bus ligero (o RabbitMQ si es necesario).
- Gestionar el versionado de endpoints de forma independiente por servicio.

Fase 3: Contenerización y despliegue distribuido

Objetivo: Facilitar despliegue, escalado y aislamiento de fallos.

- Crear un Dockerfile para cada microservicio.
- docker-compose y Helm charts para ambientes locales y de staging.
- Desplegar servicios en clústeres separados (Kubernetes/ECS) según su tipo:
    - memory-services
    - handler-services
    - search-service
    - cognitive-orchestrator como orquestador central

Fase 4: Testing y validación aislada

Objetivo: Asegurar calidad sin romper integración.

- Pruebas unitarias por servicio.
- Pruebas de integración entre servicios usando mocks.
- Test de contratos entre el CognitiveOrchestrator y cada microservicio.
- Pruebas end-to-end simulando conversaciones reales.

Fase 5: Observabilidad y gobernanza

Objetivo: Facilitar trazabilidad y diagnóstico.

- Integrar OpenTelemetry o Jaeger para tracing distribuido.
- Logs estructurados con un correlador de user_id y request_id.
- Dashboards para cada microservicio con métricas clave (latencia, errores, uso de LLM).
- Alertas automáticas si algún servicio no responde o lanza errores frecuentes.

Fase 6: Estrategia de evolución y despliegue continuo

Objetivo: Mejorar de forma segura y progresiva.

- CI/CD por microservicio con rollback y blue/green deployment.
- Canary releases para probar nuevos modelos LLM o prompts sin afectar a todos los usuarios.
- A/B testing entre versiones del orquestador o handlers para evaluar mejoras.

Fase 7: Escalabilidad horizontal y multiregión

Objetivo: Adaptarse a demanda variable y usuarios globales.

- Permitir escalar microservicios de memoria según tipo de carga (e.g. Redis para uso frecuente, Mongo para persistencia).
- Replicar el search_handler y orchestrator en distintas regiones con balanceo de carga.
- Separar usuarios por región con posibilidad de failover.