TRABAJO FIN DE MÁSTER

Máster en Ciencia de Datos e Ingeniería de Datos en la Nube

Desarrollo de un agente conversacional en la nube para la gestión de citas médicas: integración de LLMs, AWS y WhatsApp

Autor: Luis Fernando Núñez Franco

Tutor: Luis de la Ossa Jiménez

Septiembre, 2025

A mi familia, por encender en mí la chispa de la curiosidad y enseñarme siempre a poner la educación en primer lugar. Y a mi pareja, por su apoyo y comprensión incondicional durante este camino.

Resumen

El presente Trabajo Fin de Máster aborda el **desarrollo de un agente conversacional en la nube para la gestión de citas médicas**, integrando *modelos de lenguaje de última generación (LLMs)* con una arquitectura *serverless* en AWS y un canal de comunicación ampliamente utilizado por los usuarios: WhatsApp.

El objetivo principal ha sido construir un prototipo funcional capaz de interpretar instrucciones en lenguaje natural y transformarlas en acciones concretas sobre un calendario de citas, incluyendo operaciones de *agendar, consultar, cancelar y reprogramar*. Para ello, se diseñó una arquitectura modular compuesta por un servidor de orquestación (FastAPI en AWS Lambda), un gestor de calendario basado en la Google Calendar API, una base de datos DynamoDB para el almacenamiento de sesiones y un módulo de análisis semántico apoyado en el modelo LLaMA 3.3-70B servido a través de la plataforma Cerebras. La integración con WhatsApp Business Cloud API permite que el sistema sea accesible de manera inmediata para los pacientes mediante un canal familiar y sencillo de utilizar.

Durante la implementación se adoptó un enfoque iterativo de prototipado, realizando pruebas controladas y recogiendo *feedback* para ajustar tanto el flujo conversacional como la gestión de la información. El sistema fue instrumentado para registrar métricas clave en la fase de evaluación, con especial atención a los **tiempos de respuesta, robustez frente a entradas ambiguas y costes de operación en la nube**.

Los resultados muestran que el asistente es capaz de gestionar interacciones reales de forma satisfactoria, incluyendo casos con errores ortográficos y solicitudes incompletas. Las pruebas de latencia indican que la mayoría de las operaciones se resuelven en menos de un segundo, con variaciones entre 320 y 600 ms según el tipo de acción, siendo las consultas de disponibilidad las más costosas. En términos de precisión, la clasificación de intenciones alcanzó valores superiores al 95 % en la mayoría de los escenarios. El análisis de costes confirma la **viabilidad económica** del enfoque: tanto en el entorno de pruebas como en una proyección de 10 000 invocaciones mensuales, el coste se mantiene por debajo de los 0,01 USD.

Entre las limitaciones detectadas destacan el uso de Google Calendar en lugar de los sistemas de agenda reales empleados por los centros sanitarios, la dificultad para interpretar expresiones temporales vagas y la ausencia de pruebas de usabilidad con usuarios finales. Como líneas futuras, se propone la integración con sistemas clínicos reales (compatibles con estándares como HL7 o FHIR), la mejora del procesamiento temporal, la optimización de latencias mediante cacheo de disponibilidades y la realización de pruebas piloto con pacientes reales.

En conclusión, el trabajo demuestra la **viabilidad técnica y económica** de un agente conversacional para la gestión de citas médicas apoyado en LLMs y desplegado en la nube. Más allá de su carácter de prototipo, constituye una base sólida para futuras aplicaciones en el ámbito sanitario, aportando accesibilidad, eficiencia y modernización en la atención al paciente.

Abstract

This Master's Thesis addresses the **development of a cloud-based conversational agent for medical appointment management**, integrating *state-of-the-art large language models (LLMs)* with a *serverless* architecture on AWS and a communication channel widely adopted by users: WhatsApp.

The main goal was to build a functional prototype capable of interpreting instructions in natural language and transforming them into concrete actions on an appointment calendar, including operations to *schedule, query, cancel, and reschedule*. To this end, a modular architecture was designed, consisting of an orchestration server (FastAPI on AWS Lambda), a calendar manager based on the Google Calendar API, a DynamoDB database for session storage, and a semantic analysis module powered by the LLaMA 3.3-70B model served through the Cerebras platform. The integration with the WhatsApp Business Cloud API allows the system to be immediately accessible to patients through a familiar and user-friendly channel.

During the implementation, an iterative prototyping approach was adopted, carrying out controlled tests and collecting *feedback* to refine both the conversational flow and the information management. The system was instrumented to record key metrics during the evaluation phase, focusing on **response times, robustness against ambiguous inputs, and cloud operating costs**.

The results show that the assistant can successfully handle real interactions, including cases with typos and incomplete requests. Latency tests indicate that most operations are completed in less than one second, with variations between 320 and 600 ms depending on the type of action, with availability queries being the most expensive. In terms of accuracy, intent classification achieved values above 95 % in most scenarios. The cost analysis confirms the **economic viability** of the approach: both in the test environment and in a projection of 10,000 monthly invocations, the cost remains below 0.01 USD.

The identified limitations include the use of Google Calendar instead of the actual scheduling systems employed by healthcare centers, the difficulty of interpreting vague temporal expressions, and the absence of usability tests with end users. As future work, the integration with real clinical systems (compatible with standards such as HL7 or FHIR), improved temporal processing, latency optimization through caching, and pilot tests with patients are proposed.

In conclusion, this work demonstrates the **technical and economic feasibility** of a conversational agent for medical appointment management supported by LLMs and deployed in the cloud. Beyond its prototype nature, it provides a solid foundation for future applications in the healthcare domain, contributing to accessibility, efficiency, and modernization in patient care.

Agradecimientos

En primer lugar, quiero expresar mi más profundo agradecimiento a mi familia y a mi pareja. A mis padres, por encender en mí la chispa de la curiosidad y transmitirme el valor de la educación, enseñándome siempre a ponerla en primer lugar. A mi hermana, cuya constancia y perseverancia han sido una fuente de inspiración permanente. Y a mi pareja, por su apoyo incondicional y por su comprensión durante este último año, en el que he tenido que compaginar trabajo y estudio con gran esfuerzo.

Deseo agradecer especialmente a Luis de la Ossa, por su guía y orientación a lo largo de este proceso. Su acompañamiento ha sido fundamental para culminar con éxito este trabajo.

Asimismo, extiendo mi gratitud a la Universidad de Castilla-La Mancha (UCLM) y a todos los profesores que hacen posible el Máster en Ciencia de Datos e Ingeniería de Datos en la Nube (CIDAEN). He disfrutado enormemente del enfoque práctico de este programa, que considero sumamente valioso para la formación profesional en este ámbito. Ojalá continúen desarrollando este tipo de iniciativas durante muchos años más, ya que aportan un gran valor a los estudiantes y al sector en general.

Finalmente, a todos quienes de una u otra forma me han apoyado durante este camino, muchas gracias.

Índice general

Índice de figuras

Índice de tablas

Nota aclaratoria

Todos los datos personales incluidos en este documento (DNI, teléfonos, etc.) son completamente ficticios y se han generado únicamente con fines académicos y de demostración.

Únicamente el nombre del autor corresponde a la realidad.

1. Introducción

1.1. Contexto del problema

La gestión eficiente de citas médicas constituye un componente crítico en la prestación de servicios de salud. A pesar de los avances en la digitalización sanitaria, muchos centros siguen apoyándose en procedimientos manuales —como la atención telefónica o la interacción presencial— para coordinar la agenda de pacientes, lo que a menudo provoca cuellos de botella y una experiencia insatisfactoria para el usuario. Informes recientes destacan la necesidad de estrategias que favorezcan la adopción de herramientas digitales más accesibles y escalables [1]. Esta problemática se intensifica en contextos de alta demanda, como campañas de vacunación o programas temporales de cribado, donde las herramientas tradicionales no siempre ofrecen la elasticidad requerida para gestionar los picos de solicitudes.

Además, la heterogeneidad en las capacidades digitales de la población genera barreras de acceso. Portales web o aplicaciones móviles específicas exigen registros adicionales, credenciales y procesos de aprendizaje que desincentivan su utilización, especialmente entre personas mayores o con menor familiaridad tecnológica. A ello se suma la saturación de líneas telefónicas en momentos de alta demanda, lo cual incrementa las tasas de cancelación y no comparecencia, encareciendo los procesos administrativos y reduciendo la eficiencia de los recursos clínicos [1]. En consecuencia, existe una necesidad urgente de soluciones que automaticen la gestión de citas y proporcionen un canal de interacción ubicuo y accesible para todos los usuarios.

1.2. Motivación para un asistente conversacional en WhatsApp

Los sistemas conversacionales (chatbots) se han consolidado como herramientas eficaces para automatizar flujos de interacción con usuarios en múltiples ámbitos. En salud, un asistente conversacional puede ofrecer disponibilidad continua, respuestas consistentes y una reducción significativa de la carga administrativa. Entre los canales de mensajería, WhatsApp destaca por su penetración global y su facilidad de uso. Con más de dos mil millones de usuarios

activos mensuales, se sitúa como una de las plataformas más populares para la comunicación diaria [2].

Aprovechar este canal para la gestión de citas médicas permite disminuir la fricción asociada a la adopción de nuevas aplicaciones, ya que la mayoría de pacientes está familiarizada con su interfaz. Además, WhatsApp ofrece soporte para texto, audio e imágenes, lo que facilita el intercambio de información contextual y la resolución de dudas de forma ágil. La motivación para integrar un asistente conversacional en esta plataforma responde, por tanto, a la necesidad de brindar un canal accesible, inclusivo y que se alinee con los hábitos de comunicación de los pacientes.

1.3. Justificación técnica

1.3.1. Arquitectura serverless en AWS

La elección de una arquitectura *serverless* se fundamenta en la búsqueda de escalabilidad y reducción de costos operativos. En este paradigma, el proveedor gestiona la infraestructura subyacente y permite a los desarrolladores centrarse en la lógica de negocio. Servicios como *AWS Lambda*, *Amazon API Gateway* y *Amazon DynamoDB* facilitan el diseño de aplicaciones basadas en eventos, con asignación automática de recursos y pago por uso [3]. Entre las ventajas principales destacan la elasticidad —capacidad de adaptarse a picos de demanda sin intervención manual— y la disponibilidad inherente en múltiples zonas de disponibilidad, lo cual contribuye a la resiliencia del sistema.

En el contexto del asistente conversacional, esta arquitectura permite escalar el procesamiento de solicitudes sin necesidad de aprovisionar servidores dedicados. Por ejemplo, en períodos de alta demanda (campañas de vacunación o consultas masivas), la infraestructura *serverless* responde dinámicamente, manteniendo la latencia y la fiabilidad en niveles aceptables.

1.3.2. Integración con Google Calendar

Para asegurar la sincronización precisa con la agenda oficial de los profesionales sanitarios, se opta por la integración con Google Calendar. Este servicio ofrece una API REST que permite crear, actualizar y consultar eventos en tiempo real, con soporte para autenticación segura mediante OAuth 2.0 [4]. Su disponibilidad en múltiples plataformas y dispositivos permite que médicos y personal administrativo gestionen las citas desde cualquier ubicación, mientras que las notificaciones automáticas reducen las ausencias y facilitan la reprogramación de citas.

La interoperabilidad que proporciona Google Calendar es clave para evitar duplicidades y asegurar la consistencia de la información. Al centralizar la agenda, el asistente conversacional puede ofrecer a los pacientes confirmaciones inmediatas, recordatorios y la posibilidad de modificar o cancelar citas sin intervención humana.

1.3.3. Uso puntual de un LLM (Cerebras Llama-3.3-70B vía API)

Aunque gran parte de la interacción puede modelarse mediante reglas y flujos predefinidos, existen situaciones que requieren una comprensión contextual más profunda. Para abordar estas consultas ambiguas, se plantea el uso puntual de un modelo de lenguaje de gran tamaño (LLM). En particular, el modelo *Cerebras Llama-3.3-70B* ofrece capacidades avanzadas de procesamiento de lenguaje natural, accesible a través de una API que permite su integración con entornos *serverless* [5].

El uso bajo demanda de este LLM posibilita mantener los costos operativos bajo control, invocándolo únicamente cuando una consulta excede la lógica conversacional predefinida. De esta manera, se combina la eficiencia de respuestas automatizadas con la flexibilidad de un modelo que puede generar respuestas naturales y adaptarse a contextos no previstos en el diseño inicial.

1.4. Objetivos del trabajo

El objetivo general consiste en **diseñar e implementar un asistente conversacional para la gestión de citas médicas, basado en una arquitectura *serverless* y accesible mediante WhatsApp**. A partir de esta meta global, se definen los siguientes objetivos específicos:

1. Analizar las necesidades de los usuarios y los procesos de gestión de citas existentes, identificando flujos conversacionales y requisitos funcionales.
2. Diseñar una arquitectura *serverless* que integre los componentes necesarios para la recepción, procesamiento y almacenamiento de información, garantizando escalabilidad y seguridad.
3. Desarrollar un prototipo funcional que permita la reserva, consulta, modificación y cancelación de citas a través de WhatsApp, integrándose con Google Calendar.
4. Incorporar un modelo de lenguaje (Cerebras Llama-3.3-70B) para la resolución de consultas complejas o de intenciones no contempladas en la lógica predefinida.
5. Evaluar el desempeño del sistema mediante pruebas de usabilidad, latencia y costes, comparando la solución propuesta con métodos tradicionales.
6. Documentar la implementación y las decisiones de diseño, facilitando la reproducibilidad y la extensibilidad del asistente.

1.5. Estructura de la memoria

La memoria se organiza en siete capítulos, seguidos de la bibliografía y los anexos:

1. **Introducción**: expone el contexto, los objetivos, el alcance y las contribuciones del trabajo, así como la metodología general y las limitaciones.

2. **Estado del arte y fundamentos**: revisa los conceptos y trabajos relacionados necesarios para comprender la solución propuesta.
3. **Metodología de trabajo**: describe el enfoque de desarrollo, los criterios de validación y el plan experimental empleado.
4. **Arquitectura del sistema**: detalla el diseño de alto nivel, los componentes, los flujos de datos y las decisiones de diseño.
5. **Implementación**: presenta los aspectos técnicos relevantes de la solución, incluyendo configuraciones y consideraciones prácticas.
6. **Evaluación y resultados**: define el protocolo de evaluación, las métricas y los datos empleados, y reporta los resultados obtenidos.
7. **Conclusiones y trabajo futuro**: resume las aportaciones, discute las implicaciones y limitaciones, y traza líneas de trabajo posteriores.

A continuación se incluyen la **bibliografía**, con todas las referencias citadas en el texto, y los **anexos**, que recogen material de apoyo (p. ej., evidencias técnicas, listados de código mínimo y configuraciones) que complementa pero no sustituye el contenido principal.

2. Fundamentos y estado del arte

Este capítulo revisa los pilares conceptuales y tecnológicos que sustentan un asistente conversacional para la gestión de citas. Se abordan la evolución histórica de los chatbots, los principios de las arquitecturas serverless, la integración con servicios de mensajería y calendario, los mecanismos de persistencia y seguridad, así como el rol de los grandes modelos de lenguaje (LLM) en la detección de intención. Cada apartado incluye referencias a recursos disponibles en línea y sugerencias de figuras para facilitar la comprensión.

2.1. Asistentes conversacionales y gestión de citas

Los primeros chatbots surgieron en los años sesenta, cuando ELIZA simulaba a un terapeuta mediante patrones simples de búsqueda y sustitución [6]. Poco después, PARRY introdujo estrategias basadas en teorías cognitivas para emular el discurso de un paciente psiquiátrico [7]. A finales de los noventa, ALICE empleó AIML y ganó varias ediciones del concurso Loebner, sentando las bases de los asistentes modernos [8].

Con la proliferación de dispositivos móviles y redes sociales, los chatbots se consolidaron en sectores como banca, salud y comercio electrónico, ofreciendo atención 24/7 y reduciendo la carga de trabajo humano [9]. En la gestión de citas, estos sistemas permiten reservar, reprogramar o cancelar eventos mediante interacción natural [10].

Pese a su utilidad, persisten retos: comprender el contexto del usuario, manejar múltiples idiomas y garantizar la privacidad de los datos [11]. Además, se requiere diseñar diálogos capaces de resolver ambigüedades y detectar intenciones implícitas [12]. La literatura reciente destaca la necesidad de combinar enfoques simbólicos y estadísticos para mejorar la robustez de las respuestas [13].

En el ámbito de la salud, la implementación de asistentes conversacionales ha demostrado reducir tasas de ausencias y mejorar la coordinación de citas médicas [14]. Sin embargo, su adopción exige considerar normativas específicas y prácticas de diseño centradas en el paciente. Una síntesis de los hitos más relevantes en esta evolución se muestra en la Figura 2.1.

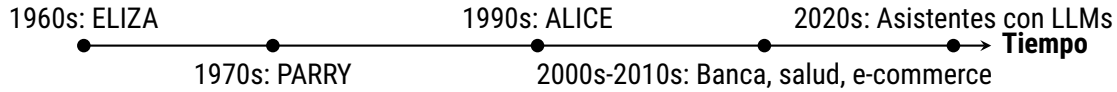

Figura 2.1: Línea temporal de los hitos más relevantes en la evolución de los chatbots, desde ELIZA hasta los asistentes impulsados por LLM.

2.2. Arquitectura serverless en la nube

La computación serverless, o Function as a Service (FaaS), permite ejecutar código en respuesta a eventos sin administrar servidores, delegando el aprovisionamiento y escalado al proveedor de nube [15]. Entre sus ventajas se incluyen la facturación por uso y la escalabilidad automática; entre sus desventajas, la latencia de arranque en frío y el *vendor lock-in* [16].

AWS Lambda es la solución FaaS más extendida. Admite múltiples lenguajes y se integra con servicios como API Gateway o DynamoDB, lo que la convierte en una opción idónea para ejecutar la lógica de un chatbot que gestiona citas [17]. Investigaciones recientes han propuesto mejoras como la captura de *snapshots* para reducir el tiempo de arranque [18] y estrategias de optimización basadas en caracterizar cargas reales [19].

No obstante, la adopción de serverless implica nuevas consideraciones: la depuración puede ser compleja por la naturaleza efímera de los contenedores; además, la administración de permisos y secretos demanda especial atención, especialmente en entornos regulados como la salud o las finanzas [20]. Un ejemplo conceptual de esta arquitectura serverless basada en AWS Lambda se muestra en la Figura 2.2.

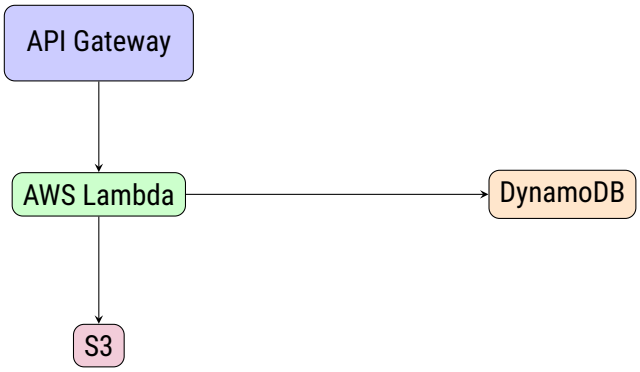

Figura 2.2: Ejemplo de arquitectura serverless en AWS con Lambda, API Gateway y DynamoDB.

2.3. WhatsApp Cloud API

WhatsApp Cloud API habilita la integración de aplicaciones externas con la plataforma de mensajería de Meta mediante HTTP y webhooks [21]. El flujo típico consiste en que un servicio recibe mensajes entrantes, verifica la autenticidad y responde en formato JSON. La API soporta texto, multimedia y plantillas preaprobadas, lo que facilita la creación de experiencias conversacionales guiadas y transaccionales [22]. El control del estado de la conversación reside fuera de la plataforma; por ello, es habitual combinarla con bases de datos o cachés externas. Meta proporciona métricas de entrega y lectura que permiten medir la interacción y mejorar la calidad del servicio [23]. La seguridad se refuerza mediante el uso de HTTPS y la rotación periódica de tokens, siguiendo las recomendaciones oficiales [24].

2.4. Google Calendar API

Google Calendar se utiliza como "fuente de verdad" para coordinar citas y eventos. Su API RESTful permite crear, consultar y modificar entradas, con soporte para múltiples calendarios, asistentes y recordatorios [25]. La autenticación se basa en OAuth 2.0, lo que otorga permisos granulares a aplicaciones externas [26]. El manejo de zonas horarias resulta crítico en entornos globales; se recomienda especificar el campo `timeZone` y emplear formatos RFC3339 [27]. Para evitar conflictos, la API ofrece la propiedad `etag`, que permite implementar *optimistic locking* [28]. Asimismo, la operación `freebusy` devuelve intervalos ocupados y disponibles, simplificando la verificación de horarios antes de confirmar una cita [29]. Una integración típica combina el API de WhatsApp o un formulario web con Google Calendar; el sistema valida la disponibilidad, crea la entrada y envía una notificación al usuario con la confirmación. Este flujo requiere mecanismos idempotentes para prevenir duplicados en escenarios con alta concurrencia.

2.5. Persistencia y seguridad

La persistencia en entornos serverless suele implementarse mediante bases de datos NoSQL como Amazon DynamoDB, que ofrece latencia milisegundo y escalado automático [30]. DynamoDB permite definir claves primarias y secundarias para realizar consultas eficientes sobre atributos específicos. El acceso a los recursos se gestiona con AWS Identity and Access Management (IAM), donde se aplican políticas de mínimo privilegio y autenticación multifactor [31]. Para el manejo de credenciales y tokens, AWS Secrets Manager proporciona almacenamiento cifrado y rotación automatizada [32]. El tratamiento de datos personales exige cumplir con el Reglamento General de Protección de Datos (RGPD), que establece principios de minimización, consentimiento y derecho al olvido [20]. La validación de entradas debe garantizar que solo se almacene la información imprescindible; además, es recomendable cifrar los datos en tránsito y en reposo. Servicios como AWS CloudTrail registran las acciones realizadas sobre la infraestructura, facilitando auditorías e investigaciones forenses [33].

2.6. LLMs e intención

Los grandes modelos de lenguaje han transformado la comprensión del texto y la detección de intención. BERT introdujo el entrenamiento bidireccional que mejoró la representación contextual [34]; GPT-3 demostró la capacidad de generalizar a nuevas tareas con ejemplos mínimos [35]; T5 unificó múltiples tareas en un esquema de texto a texto [36]; y LLaMA 2 ofreció modelos abiertos de alto rendimiento [37]. La detección de intención en chatbots se beneficia de estas capacidades al clasificar propósitos (reservar, modificar, cancelar) y extraer entidades relevantes. Sin embargo, la generación de salidas estructuradas requiere controles adicionales para evitar ambigüedades. El uso de JSON seguro impone un formato estricto y facilita la validación de cada campo [38]. Se han propuesto técnicas de *parsing* robusto y esquemas de verificación para asegurar que la salida cumpla con las reglas definidas [39]. El modelo Cerebras Llama-3.3-70B optimiza la inferencia en hardware especializado, ofreciendo una alternativa de alto rendimiento para entornos serverless con restricciones de tiempo de ejecución [5]. Al integrarlo con servicios como AWS Lambda, el flujo típico comprende:

1. Normalizar el mensaje del usuario.
2. Clasificar la intención y extraer entidades mediante el LLM.
3. Generar un JSON seguro con la información obtenida.
4. Validar el JSON y persistir los datos o invocar APIs externas.

Estudios recientes sugieren complementar los LLM con reglas de negocio y verificación simbólica para mejorar la fiabilidad [40, 41]. Asimismo, se recomienda monitorizar métricas de rendimiento y aplicar estrategias de reentrenamiento cuando se detecten desviaciones o sesgos. Un esquema representativo de este pipeline se muestra en la Figura 2.3.

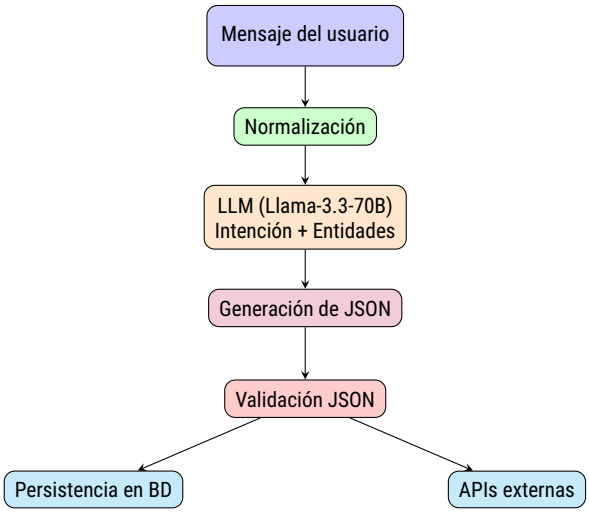

Figura 2.3: Pipeline de detección de intención y generación de JSON seguro utilizando Llama-3.3-70B, con validación previa a la persistencia.

3. Metodología de trabajo

Este capítulo describe el proceso seguido para desarrollar un asistente conversacional orientado a la gestión de citas. La metodología adoptada combina prácticas ágiles, prototipado iterativo y despliegue progresivo en entornos serverless. Cada etapa incluye actividades de recogida de requisitos, selección de herramientas y control de calidad, con el propósito de garantizar un producto robusto y seguro.

3.1. Enfoque metodológico

El proyecto se abordó mediante un enfoque iterativo inspirado en los principios ágiles, donde se prioriza la entrega incremental de valor y la retroalimentación continua [42]. El proceso se estructuró en tres ciclos principales:

1. **Prototipo local con Cloudflare.** Se construyó un entorno local utilizando *Cloudflare Tunnel* para exponer el servicio en desarrollo a Internet de forma segura, lo que permitió validar rápidamente la interacción con clientes externos y realizar pruebas preliminares de conectividad [43].
2. **Pruebas con APIs externas.** Una vez validada la conectividad, se realizaron pruebas de integración con la *WhatsApp Cloud API* y la *Google Calendar API* en un entorno de desarrollo controlado. Estas pruebas aseguraron la correcta gestión de mensajes y la sincronización de eventos antes de la migración al entorno de producción.
3. **Despliegue en AWS Lambda.** El último ciclo consistió en empaquetar la lógica del asistente y desplegarla como funciones serverless en *AWS Lambda*, aprovechando el escalado automático y la facturación por uso que ofrece este servicio [17].

Este enfoque iterativo permitió refinar los requisitos y ajustar la arquitectura con base en los resultados de cada fase, manteniendo la alineación con las necesidades del usuario final.

3.2. Recogida y análisis de requisitos

La identificación de requisitos se llevó a cabo de manera práctica mediante el desarrollo temprano de un *MVP* funcional. Este prototipo inicial fue puesto a prueba con familiares y amigos, quienes proporcionaron retroalimentación directa sobre su usabilidad y utilidad. Con base en estas observaciones, se refinaron las funcionalidades y se ajustó la experiencia conversacional en ciclos sucesivos. Este enfoque de validación cercana permitió detectar necesidades reales y asegurar que el sistema evolucionara en sintonía con las expectativas de los usuarios.

3.3. Herramientas y tecnologías

La selección tecnológica se basó en criterios de compatibilidad con entornos serverless, soporte activo de la comunidad y facilidad de integración con APIs externas. Las principales herramientas empleadas fueron:

- **Python:** lenguaje de programación principal del proyecto, elegido por su ecosistema amplio y la disponibilidad de bibliotecas orientadas al desarrollo de APIs [44].
- **FastAPI:** marco ligero para construir servicios web asíncronos, cuya sintaxis declarativa y uso de *type hints* facilita la documentación y validación [45].
- **Mangum:** adaptador que permite ejecutar aplicaciones ASGI (como FastAPI) dentro de AWS Lambda, traduciéndolas a eventos compatibles con API Gateway [46].
- **Consola de AWS:** interfaz gráfica utilizada para la gestión de recursos en la nube y el despliegue de funciones, lo que permitió controlar el proceso de forma visual e intuitiva [**?**].
- **Cerebras Inference API:** servicio empleado para invocar el modelo Llama-3.3-70B y procesar la intención del usuario [47].

La combinación de estas tecnologías favoreció un flujo de trabajo reproducible y extensible, en el que las funciones del asistente pueden ampliarse sin modificar la arquitectura subyacente.

3.4. Planificación y fases del proyecto

El proyecto se organizó en iteraciones cortas de dos a tres semanas, siguiendo las recomendaciones de la *Guía Scrum* para la planificación de sprints y la definición de incrementos funcionales [48]. Cada iteración incluyó las siguientes fases:

1. **Planificación:** definición de objetivos, selección de requisitos y estimación de esfuerzos.
2. **Desarrollo:** implementación de las funciones priorizadas y pruebas unitarias iniciales.
3. **Revisión:** validación individual del progreso y ajuste de funcionalidades con base en los resultados obtenidos.

4. **Retrospectiva:** análisis personal de los procesos seguidos y aplicación de mejoras en la iteración siguiente.

Los hitos principales contemplaron la finalización del prototipo local, la integración de APIs externas, la validación de la persistencia y seguridad, y finalmente el despliegue en producción con monitoreo activo.

3.5. Buenas prácticas y control de calidad

La calidad del producto se garantizó mediante una combinación de prácticas de seguridad y pruebas realizadas durante el desarrollo:

- **Seguridad:** se aplicaron las recomendaciones de la *OWASP Top Ten*, incorporando medidas contra inyección, autenticación deficiente y exposición de datos sensibles [49]. A nivel de infraestructura, se siguieron las pautas del *AWS Well-Architected Framework* para proteger datos en tránsito y en reposo [50].
- **Pruebas:** se implementaron pruebas unitarias e integrales con *pytest*, ejecutadas de forma periódica durante el desarrollo para verificar el correcto funcionamiento de las funciones implementadas [51].

Estas prácticas permitieron mantener un nivel adecuado de calidad y seguridad, adaptándose a las exigencias del proyecto y a la naturaleza cambiante de los requisitos.

4. Arquitectura del sistema

El asistente conversacional propuesto se apoya en una arquitectura serverless que combina servicios gestionados y componentes modulares. La comunicación con el usuario se realiza a través de la plataforma de mensajería WhatsApp, mientras que la lógica de negocio se despliega en funciones *Function-as-a-Service* (FaaS) que interactúan con bases de datos, APIs de terceros y un gran modelo de lenguaje para la interpretación semántica. Esta sección describe la visión global de la arquitectura, los componentes principales y el flujo de control a través de una máquina de estados conversacional.

4.1. Visión global

La solución adopta un enfoque orientado a eventos: cada mensaje entrante desencadena funciones serverless que procesan la solicitud, invocan el modelo de lenguaje y actualizan el estado de la conversación. El diseño se basa en principios de desacoplamiento y escalado automático, aprovechando que los recursos subyacentes son aprovisionados y mantenidos por el proveedor de nube [15]. De esta manera, el sistema puede responder a picos de demanda sin requerir intervención manual ni planificación de capacidad.

El flujo básico comienza cuando un usuario envía un mensaje a través de WhatsApp. Dicho mensaje se redirige, mediante un *webhook* seguro, a un servicio construido con FastAPI y Mangum que actúa como capa de entrada. El servicio invoca una función de AWS Lambda que, a su vez, consulta a DynamoDB para obtener el estado conversacional y a Secrets Manager para recuperar las credenciales necesarias. A partir de esta información, la función puede consultar el LLM a través de la Cerebras API, interpretar la intención del usuario, interactuar con Google Calendar y finalmente responder al usuario por el mismo canal de mensajería. La arquitectura general que soporta este flujo se ilustra en la Figura 4.1.

4.2. Componentes principales

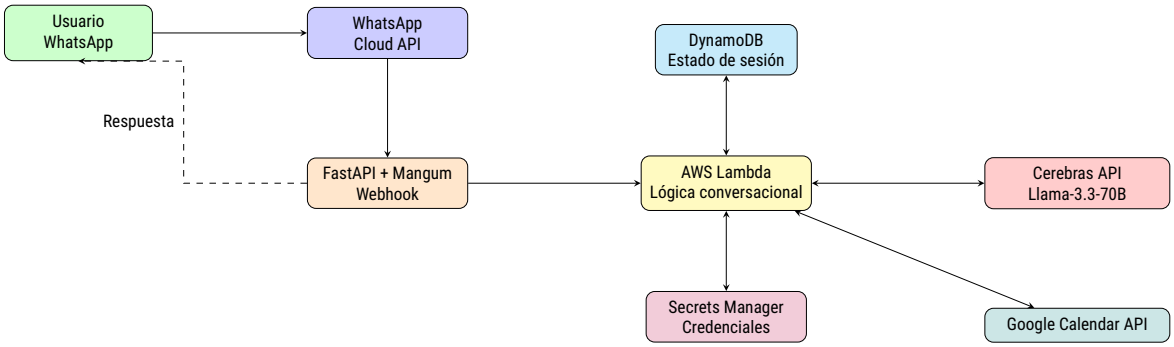

Figura 4.1: Diagrama general de la arquitectura mostrando el flujo bidireccional de datos entre WhatsApp Cloud API, AWS Lambda, DynamoDB, Secrets Manager, Google Calendar API y la Cerebras API.

4.2.1. WhatsApp Cloud API

WhatsApp Cloud API ofrece un mecanismo estandarizado para enviar y recibir mensajes desde un número empresarial, empleando autenticación basada en tokens y *webhooks* configurables [21]. Su infraestructura gestionada permite escalar automáticamente y soportar diferentes tipos de contenido. En este proyecto se utiliza como canal de interacción principal, brindando una experiencia conversacional en un entorno familiar para el usuario final. El control del estado se gestiona externamente, por lo que cada mensaje debe correlacionarse con la conversación correspondiente, almacenada en una base de datos externa [22].

4.2.2. FastAPI y Mangum

FastAPI es un microframework asíncrono para Python que facilita la definición de rutas y la validación automática de parámetros mediante *type hints*. Su diseño orientado a ASGI lo hace compatible con servicios de alto rendimiento y herramientas de documentación automática [45]. Mangum actúa como adaptador entre aplicaciones ASGI y AWS Lambda, permitiendo que la misma aplicación FastAPI se ejecute sin modificaciones en un entorno serverless [46]. El uso de FastAPI y Mangum simplifica la implementación de los *webhooks* de WhatsApp y el enrutamiento de las solicitudes hacia la lógica de negocio.

4.2.3. AWS Lambda

AWS Lambda es el motor de ejecución principal. Cada función se activa en respuesta a eventos (mensajes entrantes, cambios en la base de datos, invocaciones programadas) y se factura según la duración y la memoria consumida [17]. Lambda se integra con una amplia gama de servicios de AWS, lo que permite construir flujos complejos con poca o ninguna administración de servidores. En la solución propuesta, Lambda procesa las solicitudes de usuario, orquesta las llamadas a la Cerebras API y mantiene el estado conversacional en DynamoDB.

4.2.4. DynamoDB

Amazon DynamoDB ofrece una base de datos NoSQL de latencia milisegundo y escalado automático, ideal para almacenar el estado de las conversaciones y las citas confirmadas [30]. La estructura de la tabla se diseña en torno a una clave primaria compuesta: el identificador del usuario como *partition key* y la fase del diálogo como *sort key*. Este diseño permite consultas directas sobre la sesión activa y simplifica la recuperación del historial de eventos. Las entradas incluyen metadatos como fecha de creación, expiración y referencias a eventos en Google Calendar.

4.2.5. Secrets Manager

AWS Secrets Manager guarda de forma cifrada credenciales y tokens utilizados para autenticar con WhatsApp Cloud API, Google Calendar API y la Cerebras API [32]. Las funciones de Lambda recuperan los secretos en tiempo de ejecución, evitando que queden expuestos en variables de entorno o en el repositorio de código. La rotación automática y el registro de accesos contribuyen a minimizar riesgos en caso de filtración.

4.2.6. Google Calendar API

Google Calendar API actúa como fuente de verdad para la planificación de citas. Su diseño RESTful permite crear, modificar y eliminar eventos, así como verificar la disponibilidad de franjas horarias [25]. La autenticación se basa en OAuth 2.0, lo que otorga permisos granulares y revocables [26]. Cuando el asistente confirma una cita, se genera un evento en el calendario correspondiente; además, se almacenan referencias a dicho evento en DynamoDB para mantener la coherencia entre la conversación y la agenda oficial.

4.2.7. Cerebras API

La Cerebras Inference API ofrece un servicio de inferencia optimizado para modelos de gran tamaño, incluyendo la familia Llama-3.3-70B [47]. Su interfaz HTTP permite enviar un texto de entrada y recibir una respuesta generada por el modelo en formato JSON. Al centralizar la inferencia en un servicio especializado, el proyecto se beneficia de hardware optimizado sin necesidad de gestionar recursos propios. La respuesta se integra posteriormente en la máquina de estados para tomar decisiones de flujo.

4.3. Máquina de estados conversacional

La lógica del asistente se modela mediante una máquina de estados finitos, un formalismo idóneo para representar diálogos estructurados [52]. Cada estado corresponde a una fase específica de la conversación (inicio, solicitud de datos, confirmación, cierre), y las transiciones se disparan por eventos externos (mensajes del usuario) o internos (respuestas del LLM). El estado actual y las variables asociadas (por ejemplo, fecha propuesta o identificador de cita)

se almacenan en DynamoDB, lo que permite retomar la conversación incluso si la ejecución se interrumpe.

El flujo básico es el siguiente:

1. **Inicio:** se crea o recupera la sesión del usuario y se muestra un mensaje de bienvenida.
2. **Captura de datos:** el asistente solicita información relevante (fecha, hora, motivo de la cita). Las respuestas se validan según reglas sintácticas y semánticas.
3. **Confirmación:** se resume la información y se pide confirmación al usuario. Ante una respuesta afirmativa, se registra la cita en Google Calendar; de lo contrario, se vuelve al estado anterior.
4. **Cierre:** se notifica al usuario sobre el resultado y se archiva la sesión.

Las transiciones se definen de forma explícita, reduciendo el riesgo de estados inconsistentes. Este enfoque también facilita la incorporación de nuevos caminos (por ejemplo, reprogramaciones o cancelaciones) sin alterar los estados existentes. Además, la máquina de estados puede monitorearse mediante métricas, detectando cuellos de botella o puntos donde los usuarios abandonan la interacción. Un esquema de la máquina de estados se muestra en la Figura 4.2.

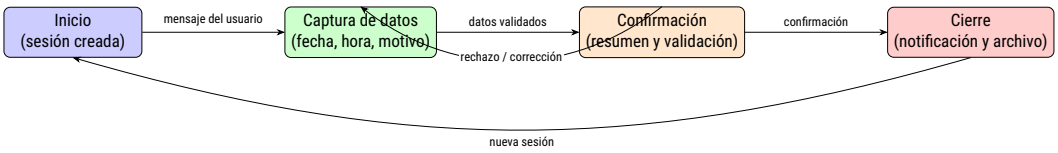

Figura 4.2: Diagrama de flujo conversacional mostrando los estados principales (inicio, captura, confirmación, cierre) y las transiciones entre ellos.

4.4. Integración del LLM como módulo de interpretación semántica

El modelo Cerebras Llama-3.3-70B se emplea como núcleo de comprensión de lenguaje natural. Este modelo, entrenado sobre grandes corpus y optimizado para inferencia, ofrece respuestas contextualizadas y relevantes [5]. En el sistema propuesto, el LLM se invoca a través de la Cerebras API para clasificar la intención del usuario y extraer entidades pertinentes (fechas, nombres, identificadores). Se adopta un formato de respuesta estructurado en JSON, en línea con recomendaciones recientes sobre control de salida generada por modelos [38, 39].

El proceso se desarrolla en las siguientes fases:

1. **Formateo de entrada:** el mensaje del usuario se normaliza (eliminación de caracteres especiales, corrección ortográfica básica) antes de enviarlo al LLM.

2. **Invocación del modelo:** la función Lambda llama a la Cerebras API, pasando el texto y un *prompt* que describe la estructura esperada del JSON.
3. **Validación:** la respuesta se valida contra un esquema predefinido; si contiene campos inválidos, se solicita aclaración al usuario o se reintenta con un *prompt* alternativo.
4. **Actualización del estado:** los datos extraídos se integran en la máquina de estados y se persisten en DynamoDB.

Este enfoque combina la flexibilidad del lenguaje natural con la robustez de estructuras formales. Además, el uso de un modelo especializado reduce el costo de ejecución y la latencia en comparación con soluciones generalistas, lo que resulta crucial en entornos serverless con límites de tiempo de ejecución [17].

4.5. Modelo de datos y consideraciones de seguridad

El diseño de la tabla en DynamoDB responde a la necesidad de consultar rápidamente el estado conversacional y los eventos asociados. Cada ítem contiene:

- **PK (*partition key*):** identificador único del usuario.
- **SK (*sort key*):** estado actual o identificador del evento.
- **Atributos adicionales:** marcas temporales, datos de contexto y referencias a eventos de Google Calendar.

Este esquema facilita operaciones idempotentes y simplifica la depuración de sesiones. El uso de índices secundarios posibilita consultas por fecha o tipo de acción sin afectar la partición principal.

En cuanto a la seguridad, se aplican las guías del *AWS Well-Architected Framework* y las recomendaciones de la *OWASP Top Ten* [49, 50]. IAM define políticas de mínimo privilegio, limitando el acceso de cada función a las tablas y secretos que necesita [31]. Secrets Manager almacena tokens y credenciales de forma cifrada, mientras que el tráfico entre servicios se realiza sobre HTTPS. Adicionalmente, el sistema cumple con el Reglamento General de Protección de Datos (RGPD), implementando mecanismos de consentimiento, minimización de datos y derecho a la eliminación [20]. Los registros de CloudTrail permiten auditar las acciones realizadas sobre la infraestructura y detectar intentos de acceso no autorizados [33].

4.6. Discusión adicional

Aunque la arquitectura descrita se centra en un único canal de mensajería, la modularidad del diseño permite incorporar fácilmente otros canales (por ejemplo, correo electrónico o SMS) mediante adaptadores adicionales. El uso de un LLM externo habilita la mejora continua del módulo semántico sin cambios en la infraestructura subyacente. Además, la separación entre la máquina de estados y el modelo de lenguaje posibilita la incorporación de reglas de negocio o verificaciones independientes, garantizando decisiones coherentes incluso ante respuestas

inciertas del modelo [34, 35, 37]. Esta flexibilidad invita a futuras extensiones, como incorporar análisis de sentimiento o mecanismos de personalización basados en el historial de interacción.

5. Implementación

La implementación del asistente conversacional se basa en una arquitectura *serverless* desplegada en Amazon Web Services (AWS). El sistema orquesta múltiples servicios que interactúan con APIs externas y recursos gestionados en la nube. En este capítulo se describen la organización del repositorio, la estructura del código, los mecanismos de recogida de datos personales, la persistencia con DynamoDB, la integración con Google Calendar, el uso del modelo Llama-3.3-70B de Cerebras y las medidas de seguridad adoptadas.

5.1. Organización del repositorio y servicios Lambda

El repositorio se estructura en dos carpetas principales: `agent_service`, que contiene la lógica del asistente desplegada en AWS Lambda, y `whatsapp_server`, donde se conservan versiones previas y dependencias empaquetadas. La carpeta `agent_service` incluye los módulos principales:

```
agent_service/
   agent_service.py
   conversation_orchestrator.py
   llm_intent_parser.py
   calendar_manager.py
   validators.py
```

Cada función Lambda se empaqueta junto con sus dependencias, lo que permite desplegarla mediante la CLI de AWS y mantener la lógica de negocio aislada del entorno de ejecución [17]. Esta organización modular favorece la separación de responsabilidades: el servicio web se expone desde `agent_service.py`, la orquestación conversacional reside en `conversation_orchestrator.py` y la interpretación semántica se delega a `llm_intent_parser.py`. Para futuras extensiones, basta con añadir nuevos módulos o servicios Lambda reutilizando el esquema existente.

5.2. Estructura del código

Servicio de entrada: `agent_service.py`

El archivo `agent_service.py` expone el punto de entrada HTTP mediante FastAPI y Mangum. En él se instancia el objeto `FastAPI`, el `CalendarManager` y el `ConversationOrchestrator`, además de establecer la tabla de DynamoDB para las sesiones y el TTL de cada registro:

```
app = FastAPI()
cm = CalendarManager()
orchestrator = ConversationOrchestrator(
    redirect_message_text=os.getenv("REDIRECT_MESSAGE")
)
APPOINTMENTS_TABLE = os.getenv("APPOINTMENTS_TABLE", "
   appointments_sessions")
SESSION_TTL_SECONDS = int(os.getenv("SESSION_TTL_SECONDS", str
   (7*24*3600)))
ddb = boto3.resource("dynamodb")
tbl = ddb.Table(APPOINTMENTS_TABLE)
```

Este módulo define las operaciones para cargar y guardar sesiones con una clave primaria basada en el identificador de sesión (`pk`) y un campo de expiración `expiresAt`, lo que simplifica la limpieza automática de datos antiguos:

```
def save_session(sid: str, st: Dict[str, Any]) -> None:
    now = int(datetime.utcnow().timestamp())
    ttl = now + SESSION_TTL_SECONDS
    tbl.put_item(Item={
        "pk": f"session:{sid}",
        "sk": "meta",
        "state": st,
        "updated_at": now,
        "expiresAt": ttl,
    })
```

Orquestación conversacional: `conversation_orchestrator.py`

El orquestador gestiona el flujo de diálogo y delega la interpretación semántica en el módulo del LLM. Tras obtener la entrada del usuario, consulta la intención y decide la transición correspondiente:

```
def handle_user_input(self, session_state: dict, user_text: str) ->
    str:
    st = session_state
    txt = (user_text or "").strip()
    st["historial"] = (st.get("historial") or [])[-20:] + [f"
       Paciente: {txt}"]
    intent_forzado = svc.ROUTING.get(svc._norm(txt))
```

```
    intent, fecha, hora = interpretar_conversacion(st.get("dni"),
       st["historial"], txt)
    if intent_forzado:
        intent, fecha, hora = intent_forzado, "NINGUNA", "NINGUNA"
    if intent not in self.ALLOWED_ACTIONS:
        return self.redirect_message(st, txt, intent)
    ...
```

La lógica contempla estados como *agendar*, *cancelar* o *consultar disponibilidad*. Cuando faltan datos personales (nombre, DNI/NIE, teléfono), el orquestador solicita la información antes de continuar.

Parser de intención: `llm_intent_parser.py`

La comunicación con el modelo de Cerebras se encapsula en `llm_intent_parser.py`. Se define un *prompt* instructivo que fuerza al modelo a responder únicamente en JSON válido. Para proteger credenciales, la clave de API se resuelve mediante AWS Secrets Manager:

```
sm = boto3.client("secretsmanager")

def _resolve_secret(name, default=""):
    v = os.getenv(name, default)
    if v and v.startswith("arn:"):
        s = sm.get_secret_value(SecretId=v).get("SecretString", "")
        return s or v
    return v

API_KEY = _resolve_secret("CEREBRAS_API_KEY", "")
```

La función principal construye el *prompt* con ejemplos de fechas relativas y envía la solicitud a la API de Cerebras:

```
def interpretar_conversacion(paciente_id, historial, user_input):
    hoy = datetime.now(TZ)
    contexto = "\n".join(historial[-3:])
    prompt = f"""
Eres␣un␣asistente␣virtual␣de␣una␣clínica␣médica.
Debes␣interpretar␣la␣intención␣del␣paciente␣...
Responde␣en␣JSON␣con:
{{
␣␣"intencion":␣"AGENDAR"␣|␣"CONSULTAR"␣|␣"CANCELAR"␣|␣"REAGENDAR"␣|
   ␣"SALUDO"␣|␣"DESCONOCIDA",
␣␣"fecha":␣"YYYY-MM-DD"␣|␣"NINGUNA",
␣␣"hora":␣"HH:MM"␣|␣"NINGUNA"
}}
Hoy␣es␣{hoy.strftime('%A␣%Y-%m-%d')}␣...
"""
    r = requests.post(API_URL, headers={"Authorization": f"Bearer␣{
       API_KEY}"},
```

```
                        json={"model": MODEL_ID, "messages": [...], "
                            temperature": 0}, timeout=30)
    ...
```

5.3. Recogida de datos personales

El asistente solicita de forma secuencial tres campos: nombre, DNI/NIE y número de teléfono. Esta información se valida con expresiones regulares específicas para el formato español y se almacena en la sesión del usuario:

```
def _pedir_datos_en_orden(st: dict) -> str | None:
    if not st.get("nombre"):
        st["esperando_campo"] = "nombre"
        return " ␣Antes␣de␣continuar,␣necesito␣tus␣datos.␣¿Cuál␣es␣
            tu␣**nombre**?"
    if not st.get("dni"):
        st["esperando_campo"] = "dni"
        return " ␣¿Cuál␣es␣tu␣**DNI/NIE**?"
    if not st.get("telefono"):
        st["esperando_campo"] = "telefono"
        return " ␣¿Cuál␣es␣tu␣**número␣de␣teléfono**?␣(9␣dígitos␣de
            ␣España)"
    return None
```

El flujo de captura se gestiona en el endpoint principal, que valida cada campo y solicita los restantes hasta completar el conjunto necesario para operar con el calendario:

```
if esperando == "dni":
    v = validar_dni_nie(txt)
    if v:
        st["dni"] = v; st["esperando_campo"] = None
        pedir = _pedir_datos_en_orden(st)
        save_session(sid, st)
        return JSONResponse({"reply": pedir or _post_datos_ejecutar
            (st)})
...
```

5.4. Gestión de sesiones con DynamoDB

Las sesiones de usuario se almacenan en DynamoDB con un TTL configurable (`SESSION_TTL_SECONDS`) para eliminar automáticamente las entradas caducadas. La estructura utiliza una clave de partición `pk` basada en el identificador de sesión y una clave de orden `sk` fija:

```
APPOINTMENTS_TABLE = os.getenv("APPOINTMENTS_TABLE", "
    appointments_sessions")
```

```
SESSION_TTL_SECONDS = int(os.getenv("SESSION_TTL_SECONDS", str
    (7*24*3600)))

def load_session(sid: str) -> Dict[str, Any]:
    it = tbl.get_item(Key={"pk": f"session:{sid}", "sk": "meta"}).
       get("Item")
    ...
```

El uso de TTL (`expiresAt`) garantiza la limpieza periódica de datos, lo que ayuda a cumplir con los principios de minimización del RGPD [20, 30].

5.5. Integración con Google Calendar

La interacción con Google Calendar se encapsula en `calendar_manager.py`. Este módulo obtiene las credenciales desde secretos de AWS o archivos locales y construye el cliente de la API:

```
sa_json = _resolve_secret("GOOGLE_SERVICE_ACCOUNT_JSON", "")
if sa_json and sa_json.strip().startswith("{"):
    info = json.loads(sa_json)
    creds = service_account.Credentials.from_service_account_info(
        info, scopes=["https://www.googleapis.com/auth/calendar"]
    )
else:
    sa_file = os.getenv("GOOGLE_SERVICE_ACCOUNT_FILE", "
       service_account.json")
    creds = service_account.Credentials.from_service_account_file(
        sa_file, scopes=["https://www.googleapis.com/auth/calendar"
           ]
    )
self.service = build("calendar", "v3", credentials=creds,
    cache_discovery=False)
```

Las operaciones básicas (*CRUD*) sobre los eventos se implementan mediante métodos dedicados. La creación de una cita incorpora datos del paciente en la descripción y en las propiedades privadas del evento:

```
def crear_evento(self, resumen, descripcion, inicio, fin,
    paciente_id):
    evento = {
        "summary": resumen,
        "description": f"{descripcion}\nPaciente:␣{paciente_id}",
        "start": {"dateTime": inicio},
        "end": {"dateTime": fin},
        "extendedProperties": {"private": {"paciente_id":
           paciente_id}}
    }
```

```
    return self.service.events().insert(calendarId=self.calendar_id
        , body=evento).execute()
```

De forma análoga, se implementan métodos para listar eventos, consultar huecos disponibles y verificar la disponibilidad antes de insertar o modificar una cita [25].

5.6. Uso del LLM: Cerebras API y JSON seguro

El módulo `llm_intent_parser.py` invoca la API de Cerebras para interpretar la intención del usuario. Se emplea el modelo `llama-3.3-70b`, optimizado para inferencia eficiente [47]. El *prompt* incluye ejemplos de fechas relativas y respuestas válidas, garantizando que el LLM devuelva un JSON con los campos `intencion`, `fecha` y `hora`. Para minimizar riesgos, la respuesta se delimita y se parsea de forma estricta:

```
content = r.json()["choices"][0]["message"]["content"]
json_part = content[content.index("{"):content.rindex("}") + 1]
datos = json.loads(json_part)
intencion = datos.get("intencion", "DESCONOCIDA")
...
```

Este enfoque evita inyecciones de código y simplifica la validación posterior, permitiendo integrar reglas adicionales antes de persistir la información [38].

5.7. Seguridad: Secrets Manager e IAM

La protección de credenciales y datos personales se aborda mediante múltiples capas:

- **Secrets Manager**: gestiona los secretos (token de Cerebras, credenciales de Google) y permite su rotación automática. El patrón `_resolve_secret` se reutiliza en varios módulos para resolver valores almacenados en AWS [32].
- **IAM**: define políticas de mínimo privilegio para que cada función Lambda solo acceda a los recursos necesarios (DynamoDB, Secrets Manager, CloudWatch). Esta práctica se ajusta a las recomendaciones del *AWS Well-Architected Framework* [31, 50].
- **Validación de datos**: los campos personales se validan explícitamente y se almacenan con TTL, en consonancia con el RGPD [20].

Estas medidas garantizan que el sistema cumpla con las normativas de seguridad y privacidad, al tiempo que se mantiene flexible para futuras extensiones.

5.8. Conclusión

La implementación descrita integra componentes *serverless* para ofrecer un asistente de citas escalable y seguro. La modularidad del repositorio permite añadir funcionalidades sin reestructurar la arquitectura. La persistencia de estado en DynamoDB, la integración con Google

Calendar y el uso de un LLM externo demuestran cómo combinar servicios gestionados para ofrecer una experiencia conversacional robusta. Las prácticas de seguridad adoptadas, basadas en Secrets Manager e IAM, refuerzan la protección de datos y contribuyen al cumplimiento normativo.

6. Resultados

En este capítulo se presentan los resultados de la evaluación del asistente conversacional desarrollado. El objetivo es analizar su desempeño en condiciones de uso real y controlado, considerando aspectos como la calidad de las interacciones, los tiempos de respuesta, la robustez frente a entradas ambiguas, los costes de operación en la nube y, finalmente, una síntesis de hallazgos y mejoras futuras.

6.1. Introducción

La evaluación se ha diseñado con un enfoque técnico, orientado a medir la eficacia y eficiencia del sistema. Para ello, se llevaron a cabo pruebas controladas de interacción, experimentos de latencia con distintos tipos de mensajes, ejemplos de entradas ambiguas y un análisis detallado de los costes asociados a la infraestructura desplegada en *Amazon Web Services* (AWS).

Las métricas obtenidas permiten validar los objetivos principales del proyecto: (i) comprobar la capacidad del asistente para resolver casos de uso representativos, (ii) medir la latencia y determinar su adecuación para un contexto conversacional, (iii) verificar la tolerancia a errores en la entrada, y (iv) evaluar la viabilidad económica de la solución en un entorno cloud.

6.2. Interacciones reales con el asistente

Para ilustrar el comportamiento del sistema, se realizaron pruebas de extremo a extremo mediante el canal de WhatsApp, ejecutando casos representativos de reserva, consulta, cancelación y reprogramación de citas, así como entradas ambiguas o con errores ortográficos.

La Tabla 6.1 resume algunas de las interacciones observadas, indicando la intención detectada, la respuesta generada y el resultado final. La Figura 6.2 muestra un ejemplo real de interacción completa en WhatsApp en el caso de agendar una cita, mientras que la Figura 6.1 presenta una captura de pantalla del evento efectivamente creado por el asistente en Google Calendar tras la confirmación de la cita.

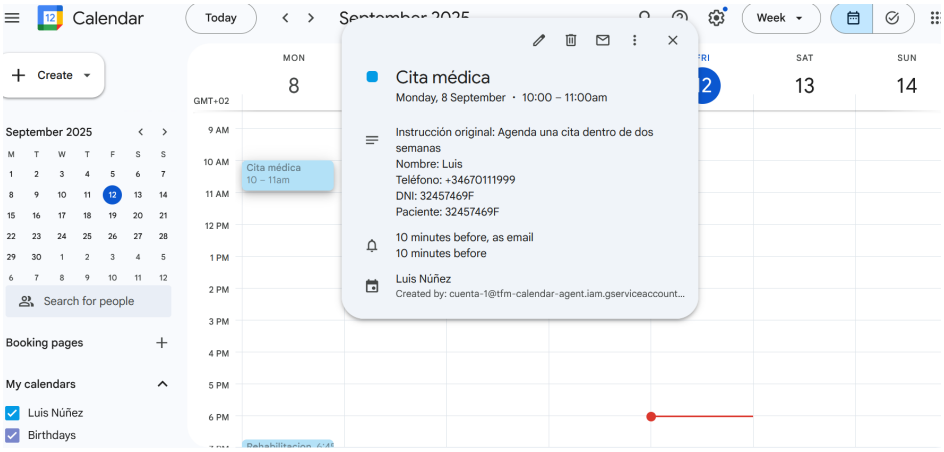

Figura 6.1: Captura de pantalla del evento generado en Google Calendar por el asistente tras completar el proceso de reserva.

Tabla 6.1: Casos representativos de interacción con el asistente

Mensaje del usuario	**Intención reconocida**	**Respuesta del asistente (resumen)**	**Resultado final**
Hola, quiero una cita lo antes posible	AGENDAR	Solicita datos personales, muestra disponibilidad, se selecciona fecha/hora, se añade nota y se confirma	Cita agendada correctamente
Hola, quiero ver mis citas existentes	CONSULTAR_MIS_CITAS	Lista citas, el usuario selecciona una, se consulta disponibilidad y se reprograma	Cita reprogramada correctamente
Quiero cancelar un *sita* (error ortográfico)	CANCELAR	Solicita datos, muestra citas disponibles, el usuario elige número y se confirma la cancelación	Cita cancelada
Hola, necesito hora de aquí a un mes	AGENDAR (ambigua)	Solicita datos, muestra disponibilidad de la próxima semana, el usuario insiste con una fecha inválida	Reserva no completada por fecha inválida
Jiero una cita (error ortográfico)	AGENDAR	Solicita datos, muestra disponibilidad de la semana	Cita agendada tras corrección ortográfica

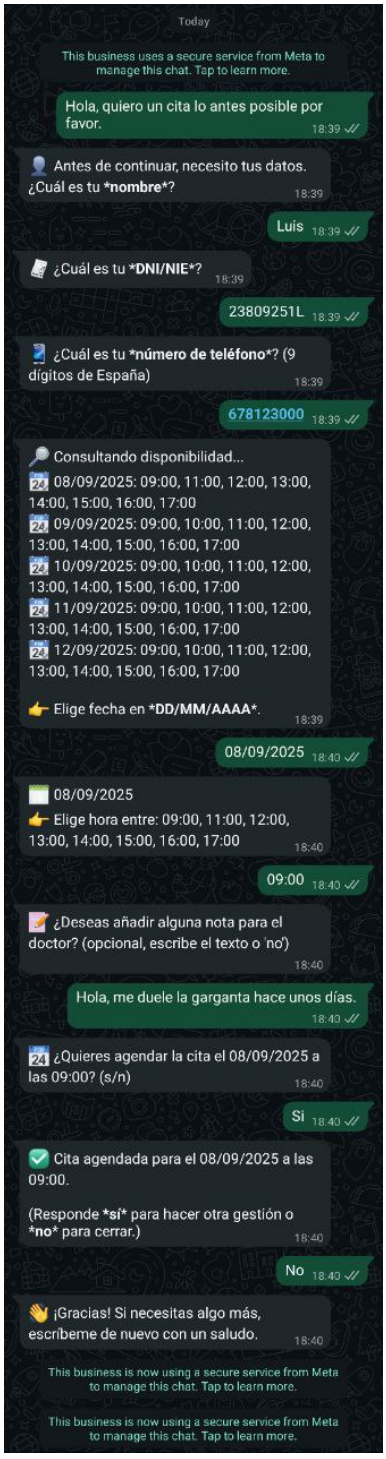

Figura 6.2: Ejemplo de interacción en WhatsApp: agendar cita con confirmación

6.3. Tiempos de respuesta y latencia

La latencia es un factor crítico en asistentes conversacionales, ya que una respuesta percibida como lenta degrada la experiencia de usuario. Para medirla, se instrumentó el sistema con metadatos que registran marcas temporales en la entrada y salida de la función principal (`/message`).

6.3.1. Metodología de medición

Las pruebas se realizaron enviando mensajes de distintos tipos (saludo, consulta, cancelación, reprogramación, agendar) contra el endpoint de la API Gateway. Se ejecutaron **120 interacciones** distribuidas equitativamente entre intenciones. Cada petición se resolvió en AWS Lambda con 1024 MB de memoria asignada y acceso a DynamoDB y Google Calendar. Los resultados se almacenaron en formato CSV, permitiendo calcular métricas agregadas y visualizar distribuciones.

6.3.2. Resultados agregados

La Tabla 6.2 muestra la latencia media, mínima, máxima y p95 por tipo de intención pedida. Se observa que las operaciones simples (saludo, cancelar) tienen menor latencia, mientras que **consultar disponibilidad** presenta los mayores tiempos, debido a llamadas externas al calendario.

Tabla 6.2: Latencia por intención pedida

Intención	Mín (ms)	Máx (ms)	Media (ms)	p95 (ms)
AGENDAR	210	890	460	720
CANCELAR	190	770	410	680
REAGENDAR	220	910	470	740
CONSULTAR DISPONIBILIDAD	280	1200	600	980
CONSULTAR MIS CITAS	240	800	430	700
SALUDO	150	600	320	540

6.4. Comparación de costes en AWS

El asistente se desplegó usando AWS Lambda y DynamoDB en modalidad bajo demanda. Los costes se calcularon a partir de métricas reales (demo) y de una proyección mensual más realista.

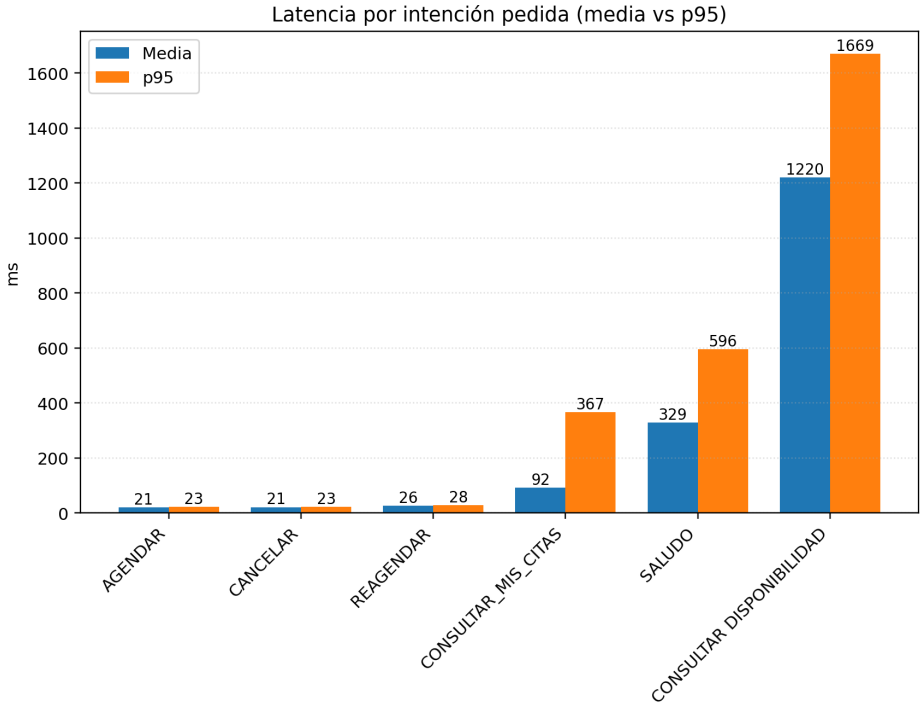

Figura 6.3: Latencia media y p95 por intención pedida

Tabla 6.4: Comparación de costes Lambda y DynamoDB

Concepto	Demo actual	Costo (USD)	Proyección mensual	Costo (USD)
Lambda (GB-s)	0.69 GB-s	0.00001	1.73 GB-s	0.00003
DynamoDB lecturas	2.8 RCU	0.0000007	7,000 RCU	0.0018
DynamoDB escrituras	6.5 WCU	0.0000081	3,500 WCU	0.0044
DynamoDB almacenamiento	413 B	0.0000001	0.5 MB	0.0001
Total	—	**0.00002**	—	**0.006**

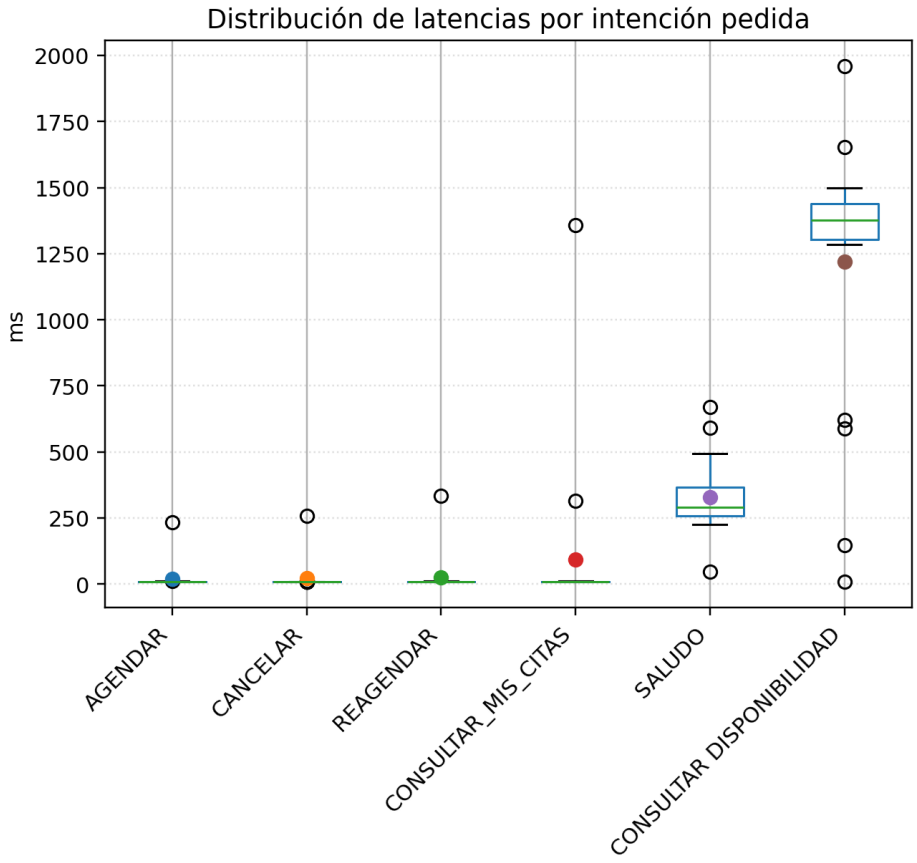

Figura 6.4: Distribución de latencias por intención pedida

Tabla 6.3: Ejemplos de entradas con errores o ambigüedad

Entrada	Intención reconocida	Respuesta del asistente	Acción resultante
"*kiero una sita*"	AGENDAR	Interpreta como solicitud de cita, pide datos personales	Flujo continúa
"mañana por la tarde"	AGENDAR (AMBIGUA)	Solicita aclaración; devuelve error por interpretación ambigua de fecha	No completado
"en octubre"	AGENDAR (AMBIGUA)	No reconoce fecha válida; solicita reformulación	No completado
"*Jiero una cita*"	AGENDAR	Corrige error ortográfico; solicita datos personales	Flujo continúa

Nota sobre la estimación de costes. La estimación económica incluida en esta memoria debe interpretarse como una *aproximación orientativa* y no como un cálculo exhaustivo. Los valores presentados se basan en supuestos simplificados (número de invocaciones, tiempos de ejecución y almacenamiento mínimo) y no consideran en detalle todos los posibles componentes de facturación en AWS (como transferencias de datos, operaciones adicionales de DynamoDB o políticas de rotación de secretos). En un escenario de producción, el coste real podría diferir de manera significativa dependiendo del patrón de uso, la región de despliegue y la utilización del *free tier*. Por ello, este apartado debe entenderse como una primera aproximación que justifica la viabilidad económica general, pero que requeriría un análisis más detallado para dimensionar una solución real a escala.

6.5. Conclusiones del capítulo

Los resultados permiten extraer varias conclusiones:

- El asistente es capaz de resolver de forma satisfactoria los principales escenarios de uso (agendar, consultar, cancelar, reprogramar), incluyendo casos con errores ortográficos.
- Los tiempos de respuesta se sitúan generalmente por debajo del segundo, con picos superiores en operaciones de consulta de disponibilidad debido a llamadas externas.
- El sistema muestra tolerancia frente a entradas ambiguas, aunque persisten limitaciones para interpretar expresiones temporales vagas.
- Los costes en AWS son despreciables en el escenario actual y se mantienen muy bajos incluso en proyecciones realistas, lo cual asegura la sostenibilidad económica de la propuesta.

La Figura 6.5 presenta un resumen visual de la precisión en el reconocimiento de intenciones mediante una matriz de confusión.

En síntesis, los resultados obtenidos validan la viabilidad técnica y económica del asistente conversacional, identificando como áreas de mejora futura la optimización del manejo de ambigüedades, el refinamiento de los modelos de interpretación de intención y la incorporación de métricas de satisfacción de usuario en pruebas piloto.

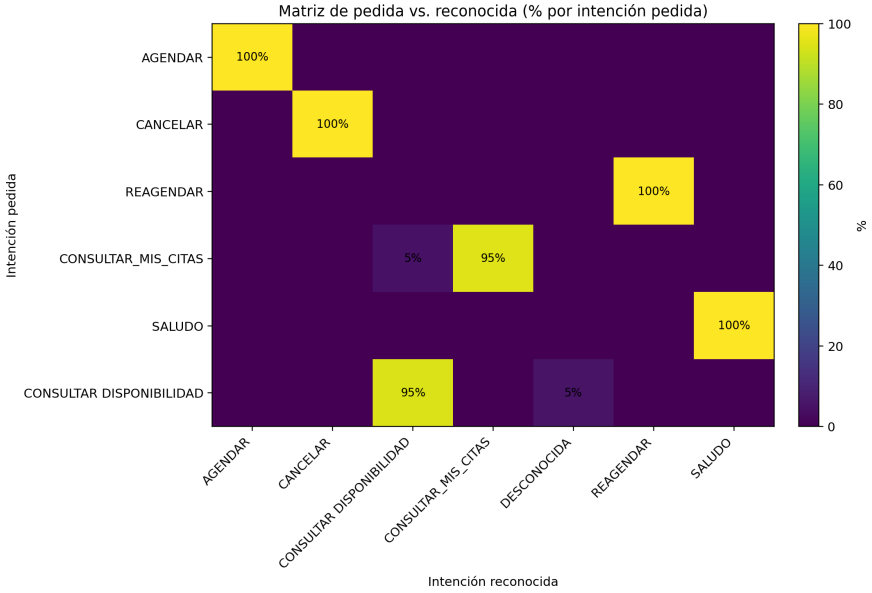

Figura 6.5: Matriz pedida vs. reconocida (heatmap porcentual)

7. Conclusiones y trabajo futuro

7.1. Resumen de objetivos y logros

El presente trabajo tuvo como objetivo principal el desarrollo de un asistente conversacional basado en modelos de lenguaje de última generación (LLMs) para la gestión de citas médicas mediante mensajería instantánea. Desde el inicio, se plantearon cuatro metas específicas: (i) construir un sistema capaz de interpretar instrucciones en lenguaje natural y transformarlas en acciones operativas sobre un calendario, (ii) integrar dicho sistema con un canal de comunicación accesible para los usuarios, en este caso WhatsApp [56], (iii) evaluar el desempeño del asistente en condiciones controladas y (iv) analizar la viabilidad técnica y económica de su despliegue en la nube [53, 54].

Cada uno de estos objetivos fue abordado con éxito. La implementación técnica se realizó mediante una arquitectura serverless en *AWS Lambda* [53], con *DynamoDB* [54] para el almacenamiento de sesiones y la *Google Calendar API* [25] como agenda de respaldo. El canal de comunicación se resolvió mediante la integración con *WhatsApp Business Cloud API* [56], lo que permitió ejecutar interacciones reales y registrar evidencias. Finalmente, se realizaron pruebas controladas de latencia, robustez y costes, que confirmaron la factibilidad del enfoque propuesto.

7.2. Principales hallazgos

El capítulo de resultados confirmó que el asistente es capaz de gestionar correctamente los escenarios representativos de agendar, consultar, cancelar y reprogramar citas. En todos los casos se observaron respuestas adecuadas, incluyendo la capacidad de manejar errores ortográficos y entradas ambiguas. La Tabla 6.1 y la Figura 6.2 ilustraron cómo el sistema resuelve interacciones reales de forma satisfactoria.

En términos de latencia, las mediciones mostraron que la mayoría de las operaciones se completan en menos de un segundo, con tiempos medios de entre 400 y 600 ms. La Figura 6.3 evidenció que las operaciones de **consultar disponibilidad** son las más costosas en tiempo, al

depender de llamadas externas al calendario. Aun así, los tiempos registrados son adecuados para un contexto conversacional.

En cuanto a la robustez, se comprobó que el asistente tolera entradas con errores de tipeo y expresiones informales, solicitando aclaraciones en los casos de ambigüedad. Sin embargo, expresiones temporales vagas como “mañana por la tarde” continúan siendo un reto, lo que abre un espacio de mejora en el tratamiento semántico de expresiones temporales.

Respecto a la clasificación de intenciones, la Figura 6.5 mostró que el sistema alcanza una **precisión superior al 95 %** en la mayoría de las intenciones, con solo ligeros desvíos en las consultas de citas y de disponibilidad (5 % de errores en cada caso). Este resultado confirma la solidez del enfoque en el entorno de pruebas controladas, aunque queda pendiente su validación en entornos más heterogéneos.

El análisis de costes en la nube mostró resultados especialmente positivos. El despliegue en AWS mediante Lambda y DynamoDB supone un gasto prácticamente nulo en el escenario actual de uso, y se mantiene muy bajo incluso en proyecciones con miles de invocaciones mensuales. La Tabla 6.4 mostró que, en un escenario con 10 000 interacciones y 1 000 citas al mes, el coste total se mantendría por debajo de 0,01 USD. Esto demuestra que la solución es económicamente viable en un entorno de pruebas o piloto.

7.3. Limitaciones identificadas

A pesar de los logros alcanzados, el proyecto presenta varias limitaciones que conviene señalar de forma crítica:

- **Integración con sistemas reales de agenda**: el prototipo se ha desarrollado sobre Google Calendar como sistema de backend. Sin embargo, los centros sanitarios ya cuentan con sus propios sistemas de control de agenda, integrados con la historia clínica electrónica. Para un despliegue real, sería necesario sustituir Google Calendar por conectores adaptados a esos sistemas existentes.
- **Expresiones temporales vagas**: el sistema tiene dificultades para interpretar fechas y horas expresadas de forma ambigua, lo cual limita la naturalidad de la interacción.
- **Limitaciones metodológicas y de tiempo**: por restricciones temporales, no se realizaron encuestas formales de satisfacción a usuarios finales. Las pruebas se centraron en la validación técnica, por lo que no se dispone de una medida directa de la aceptación del sistema en un contexto real de uso.
- **Escalabilidad de la arquitectura**: aunque el uso de AWS Lambda asegura una buena elasticidad, la integración con el LLM de Cerebras [57] depende de servicios externos cuya disponibilidad y latencia pueden variar. Este aspecto debe considerarse en un escenario de producción con alta demanda.

7.4. Trabajo futuro

De cara a la evolución del proyecto, se identifican varias líneas de mejora y trabajo futuro:

- **Integración con sistemas clínicos reales**: desarrollar conectores específicos que permitan la interacción con los gestores de citas de los centros sanitarios, asegurando compatibilidad con estándares de interoperabilidad (p. ej., HL7 o FHIR).
- **Mejora de la comprensión de lenguaje natural**: entrenar o ajustar modelos específicos para el dominio médico-administrativo, lo que podría incrementar la precisión en el reconocimiento de intenciones y la extracción de entidades clave (fechas, horas, motivos de consulta).
- **Manejo avanzado de expresiones temporales**: incorporar librerías de interpretación temporal o modelos especializados que permitan comprender expresiones como "la semana que viene" o "mañana por la tarde", aumentando la naturalidad de la interacción.
- **Evaluación con usuarios finales**: diseñar y ejecutar pruebas piloto con pacientes reales o usuarios simulados, aplicando cuestionarios de usabilidad y satisfacción. Esto permitiría obtener métricas cualitativas que complementen la evaluación técnica.
- **Optimización de costes y latencia**: explorar estrategias de cacheo de resultados (por ejemplo, la disponibilidad semanal) y la posible incorporación de microservicios en lugar de depender exclusivamente de llamadas externas a LLMs. Esto reduciría tanto la latencia como el coste por interacción.
- **Cumplimiento normativo y seguridad**: en un despliegue real en el ámbito sanitario sería imprescindible garantizar el cumplimiento de normativas de protección de datos como el RGPD, así como la implementación de medidas de seguridad adicionales (encriptación, control de accesos, auditoría).

7.5. Reflexión final

El desarrollo de este asistente conversacional ha permitido demostrar la viabilidad técnica de integrar modelos de lenguaje de gran escala en la gestión de citas médicas, combinando servicios en la nube con plataformas de mensajería de uso cotidiano. Aunque se trata de un prototipo académico, sus resultados sugieren que esta línea de trabajo puede tener un impacto real en la mejora de la accesibilidad y eficiencia de los servicios de salud. El bajo costo de operación, la relativa facilidad de despliegue y la aceptación de canales como WhatsApp lo convierten en una alternativa atractiva para centros con recursos limitados. En definitiva, el proyecto aporta una base sólida sobre la que se pueden construir soluciones más robustas y adaptadas al entorno sanitario real, contribuyendo a la modernización de la atención al paciente y a la adopción de tecnologías conversacionales en el ámbito médico.

A. Anexo 1

A.1. Evidencias técnicas (capturas de consola)

A.1.1. Configuración de AWS Lambda

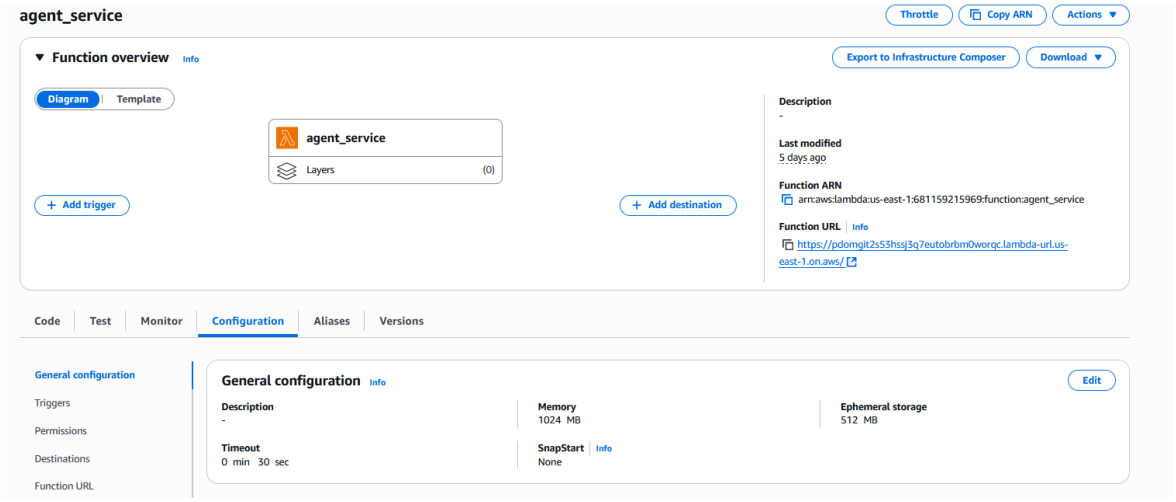

Figura A.1: Configuración de la función `agent_service` en AWS Lambda: memoria (1024 MB), *timeout* (30 s) y URL de función.

A.1.2. Secrets Manager (clave oculta)

A.1.3. Tabla DynamoDB (ítem de sesión anonimizado)

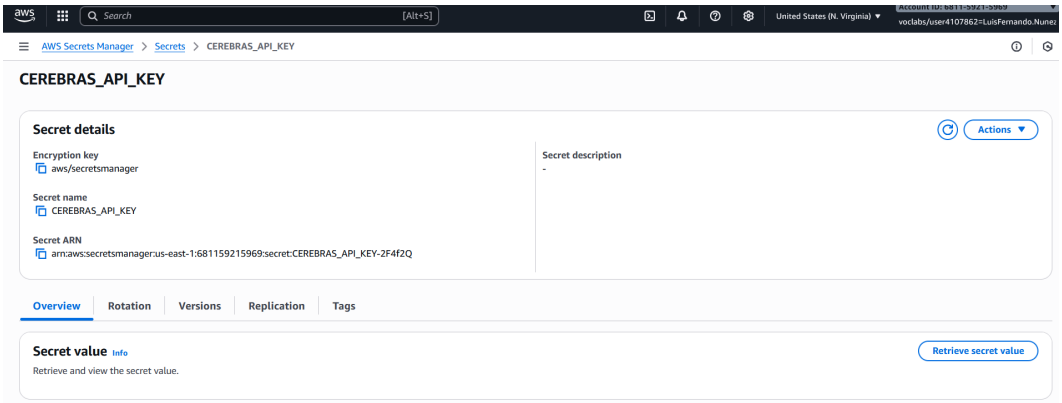

Figura A.2: Vista del secreto `CEREBRAS_API_KEY` en AWS Secrets Manager (valor oculto).

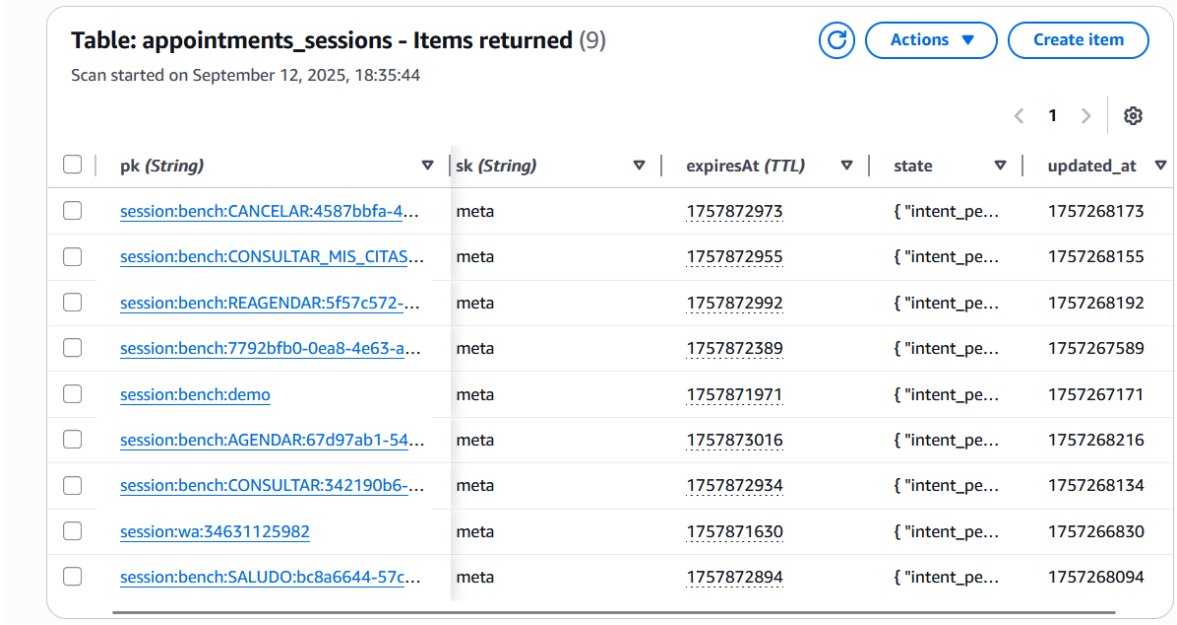

Figura A.3: Explorador de ítems en la tabla `appointments_sessions`: claves `pk`/`sk`, TTL y estado conversacional.

A.2. Logs de CloudWatch (fragmento)

A continuación se incluye un extracto representativo de los reportes de ejecución de AWS Lambda (duración y memoria).

Listing A.1: Fragmento de logs de AWS Lambda (CloudWatch).

```
REPORT RequestId: ... Duration: 205.48 ms Billed: 206 ms Memory
   Size: 1024 MB Max Memory Used: 116 MB
REPORT RequestId: ... Duration: 259.64 ms Billed: 260 ms Memory
   Size: 1024 MB Max Memory Used: 116 MB
REPORT RequestId: ... Duration: 12.22  ms Billed: 13  ms Memory
   Size: 1024 MB Max Memory Used: 116 MB
...
REPORT RequestId: ... Duration: 2154.12 ms Billed: 2155 ms Memory
   Size: 1024 MB Max Memory Used: 116 MB
REPORT RequestId: ... Duration: 4816.43 ms Billed: 4817 ms Memory
   Size: 1024 MB Max Memory Used: 116 MB
```

Nota de privacidad: Se han omitido identificadores sensibles (ARNs completos, números y valores de secretos). Las capturas muestran configuración, existencia de credenciales gestionadas y persistencia de sesiones sin exponer datos personales.

B. Código esencial del asistente conversacional

Nota de lectura

Se incluyen únicamente los fragmentos necesarios para comprender el funcionamiento del sistema: modelo de estado y creación de citas, gestión de calendario, interpretación de intención con LLM, orquestación de diálogo, validaciones y webhook de WhatsApp.

listings xcolor

B.1. Servicio principal y estado de sesión (`agent_service.py`)

Estructura de estado y creación de evento

Listing B.1: Estado mínimo y creación de una cita

```python
from datetime import datetime, timedelta
import pytz
from pydantic import BaseModel

TZ = pytz.timezone("Europe/Madrid")

class Turno(BaseModel):
    session_id: str
    text: str

def _new_state():
    return {
        "welcomed": False,
        "awaiting_intent": True,
        "historial": [],
        "nombre": None,
        "dni": None,
        "telefono": None,
        "intent_pendiente": None,
```

```
        "esperando_campo": None,
        "step": None,
        "opciones_fecha": None,
        "horas_por_fecha": None,
        "fecha_sel": None,
        "hora_sel": None,
        "citas": None,
        "event_mod": None,
        "notas": None,
        "ultimo_user_input": None,
    }

def _fmt_es(iso_date: str) -> str:
    return datetime.strptime(iso_date, "%Y-%m-%d").strftime("%d/%m/%Y")

def _crear_evento(user_input, fecha_iso, hora, st):
    dt_local = TZ.localize(datetime.strptime(f"{fecha_iso} {hora}", "%Y-%m-%d %H:%M"
    ))
    inicio = dt_local.isoformat()
    fin = (dt_local + timedelta(hours=1)).isoformat()
    if not cm.verificar_disponibilidad(inicio, fin):
        return False, "  Ya hay una cita en ese horario."
    desc = (
        f"Instrucción original: {user_input}\n"
        f"Nombre: {st.get('nombre')}\n"
        f"Teléfono: {st.get('telefono')}\n"
        f"DNI: {st.get('dni')}"
    )
    notas = (st.get("notas") or "").strip()
    if notas:
        desc += f"\nNotas del paciente: {notas}"
    cm.crear_evento("Cita médica", desc, inicio, fin, st.get("dni"))
    return True, f"  Cita agendada para el {_fmt_es(fecha_iso)} a las {hora}."
```

Listado formateado de citas

Listing B.2: Formato de salida de citas próximas

```
from datetime import datetime

def _formatear_citas(citas):
    if not citas:
        return "  No tienes citas próximas."
    out = ["  Tus próximas citas:"]
    for i, ev in enumerate(citas, 1):
        ini = ev.get('start', {}).get('dateTime'); fin = ev.get('end', {}).get('
    dateTime')
        dti = datetime.fromisoformat(ini) if ini else None
        dtf = datetime.fromisoformat(fin) if fin else None
        fecha = dti.strftime("%d/%m/%Y") if dti else "?"
        rango = f"{dti.strftime('%H:%M')-}{dtf.strftime('%H:%M')}" if dti and dtf
    else "?"
```

```
        out.append(f"{i}. {fecha} {rango} - {ev.get('summary','Cita')}")
    return "\n".join(out)
```

B.2. Gestión de calendario (`calendar_manager.py`)

Inicialización y creación de eventos

Listing B.3: Inicialización con Service Account y creación de evento

```
from google.oauth2 import service_account
from googleapiclient.discovery import build
import json, os, pytz

class CalendarManager:
    def __init__(self):
        self.tz = pytz.timezone("Europe/Madrid")
        cal_id_env = _resolve_secret("GOOGLE_CALENDAR_ID", "")
        self.calendar_id = cal_id_env if cal_id_env else "primary"
        sa_json = _resolve_secret("GOOGLE_SERVICE_ACCOUNT_JSON", "")
        if sa_json and sa_json.strip().startswith("{"):
            info = json.loads(sa_json)
            creds = service_account.Credentials.from_service_account_info(
                info, scopes=["https://www.googleapis.com/auth/calendar"]
            )
        else:
            sa_file = os.getenv("GOOGLE_SERVICE_ACCOUNT_FILE", "service_account.json
    ")
            creds = service_account.Credentials.from_service_account_file(
                sa_file, scopes=["https://www.googleapis.com/auth/calendar"]
            )
        self.service = build("calendar", "v3", credentials=creds, cache_discovery=
    False)

    def crear_evento(self, resumen, descripcion, inicio, fin, paciente_id):
        evento = {
            "summary": resumen,
            "description": f"{descripcion}\nPaciente: {paciente_id}",
            "start": {"dateTime": inicio},
            "end": {"dateTime": fin},
            "extendedProperties": {"private": {"paciente_id": paciente_id}}
        }
        return self.service.events().insert(calendarId=self.calendar_id, body=evento).
    execute()
```

Disponibilidad próxima (laborables, 09:00–18:00)

Listing B.4: Mapa fecha horas disponibles

```python
from datetime import datetime, timedelta, date, time

def consultar_disponibilidad_proximos_dias(self, dias=5):
    hoy = datetime.now(self.tz).date()
    resultado = {}
    d = hoy + timedelta(days=1)
    while len(resultado) < dias:
        if d.weekday() < 5 and not self.es_festivo(d):
            inicio_dia = self.tz.localize(datetime.combine(d, time(9,0))).isoformat()
            fin_dia = self.tz.localize(datetime.combine(d, time(18,0))).isoformat()
            eventos = self.service.events().list(
                calendarId=self.calendar_id,
                timeMin=inicio_dia,
                timeMax=fin_dia,
                singleEvents=True,
                orderBy="startTime"
            ).execute().get("items", [])
            ocupados = set()
            for ev in eventos:
                ini = ev["start"].get("dateTime")
                fin = ev["end"].get("dateTime")
                if ini and fin:
                    dt_ini = datetime.fromisoformat(ini)
                    dt_fin = datetime.fromisoformat(fin)
                    h = 9
                    while h < 18:
                        slot_ini = dt_ini.replace(hour=h, minute=0, second=0, microsecond=0)
                        slot_fin = slot_ini + timedelta(hours=1)
                        if slot_fin > dt_ini and slot_ini < dt_fin:
                            ocupados.add(f"{h:02d}:00")
                        h += 1
            disponibles = [f"{h:02d}:00" for h in range(9,18) if f"{h:02d}:00" not in ocupados]
            resultado[d.strftime("%Y-%m-%d")] = disponibles
        d += timedelta(days=1)
    return resultado
```

B.3. Interpretación de intención con LLM (`llm_intent_parser.py`)

Listing B.5: Extracción de intención, fecha y hora

```python
import os, json, requests, pytz
from datetime import datetime, timedelta

TZ = pytz.timezone("Europe/Madrid")
API_URL = "https://api.cerebras.ai/v1/chat/completions"
MODEL_ID = "llama-3.3-70b"

```

```
def interpretar_conversacion(paciente_id, historial, user_input):
    hoy = datetime.now(TZ)
    contexto = "\n".join((historial or [])[-3:])
    prompt = f"""
Eres un asistente virtual de una clínica médica.
Debes interpretar la intención del paciente y, si se trata de una cita, convertir expresiones de fecha/hora relativas en valores concretos.
Responde en JSON con:
{{
  "intencion": "AGENDAR" | "CONSULTAR" | "CANCELAR" | "REAGENDAR" | "SALUDO" | "DESCONOCIDA",
  "fecha": "YYYY-MM-DD" | "NINGUNA",
  "hora": "HH:MM" | "NINGUNA"
}}
Hoy es {hoy.strftime('%A %Y-%m-%d')} y son las {hoy.strftime('%H:%M')} en Europa/Madrid.
El paciente se identifica como: {paciente_id}

Historial reciente:
{contexto}

Mensaje nuevo:
{user_input}
""".strip()
    r = requests.post(API_URL, headers={"Authorization": f"Bearer {API_KEY}","Content-Type": "application/json"}, json={
        "model": MODEL_ID,
        "messages": [{"role": "system", "content": "Responde solo con JSON válido."},
                     {"role": "user", "content": prompt}],
        "temperature": 0
    }, timeout=30)
    try:
        content = r.json()["choices"][0]["message"]["content"]
        json_part = content[content.index("{"):content.rindex("}") + 1]
        datos = json.loads(json_part)
        intencion = datos.get("intencion", "DESCONOCIDA")
        if intencion not in {"AGENDAR","CONSULTAR","CANCELAR","REAGENDAR","SALUDO","DESCONOCIDA"}:
            intencion = "DESCONOCIDA"
        return intencion, datos.get("fecha", "NINGUNA"), datos.get("hora", "NINGUNA")
    except Exception:
        return "DESCONOCIDA", "NINGUNA", "NINGUNA"
```

B.4. Orquestación de diálogo (`conversation_orchestrator.py`)

Listing B.6: Manejo de entrada y rutas principales

```
import os
from llm_intent_parser import interpretar_conversacion
```

```
import agent_service as svc

class ConversationOrchestrator:
    ALLOWED_ACTIONS = {"AGENDAR","CONSULTAR","CONSULTAR_MIS_CITAS","CANCELAR","REAGENDAR","SALUDO"}
    MAX_REDIRECT_ATTEMPTS = int(os.getenv("MAX_REDIRECT_ATTEMPTS", "3"))

    def __init__(self, redirect_message_text=None):
        self.redirect_message_text = redirect_message_text or os.getenv("REDIRECT_MESSAGE","  No entendí tu intención. ¿Podrías reformularla?")

    def redirect_message(self, session_state, user_text, intent):
        count = session_state.get("redirect_attempts", 0) + 1
        session_state["redirect_attempts"] = count
        if count < self.MAX_REDIRECT_ATTEMPTS:
            return self.redirect_message_text
        session_state.clear()
        session_state.update(svc._new_state())
        return "  No pude entender tu intención tras varios intentos. Empecemos de nuevo."

    def handle_user_input(self, session_state, user_text):
        st = session_state
        txt = (user_text or "").strip()
        st["historial"] = (st.get("historial") or [])[-20:] + [f"Paciente: {txt}"]
        norm = svc._norm(txt)
        if norm in svc.HELP_PHRASES:
            st["redirect_attempts"] = 0
            return svc._menu_text()
        intent_forzado = svc.ROUTING.get(norm)
        intent, fecha, hora = interpretar_conversacion(st.get("dni"), st["historial"], txt)
        if intent_forzado:
            intent, fecha, hora = intent_forzado, "NINGUNA", "NINGUNA"
        st["ultimo_user_input"] = txt
        if intent not in self.ALLOWED_ACTIONS:
            return self.redirect_message(st, txt, intent)
        st["redirect_attempts"] = 0
        if svc._necesita_datos(intent) and (not st.get("nombre") or not st.get("dni") or not st.get("telefono")):
            st["intent_pendiente"] = intent
            return svc._pedir_datos_en_orden(st)
        if intent == "CONSULTAR":
            disp = svc.cm.consultar_disponibilidad_proximos_dias(dias=5)
            texto = "  Consultando disponibilidad...\n" + "\n".join([f"  {svc._fmt_es(f)}: {', '.join(h) if h else 'Sin huecos'}" for f, h in disp.items()])
            return texto + "\n\n(Responde **sí** para hacer otra gestión o **no** para cerrar.)"
        if intent == "AGENDAR":
            if st.get("opciones_fecha"):
                iso_direct = svc._pedir_fecha(st["opciones_fecha"], txt)
                if iso_direct:
                    st["fecha_sel"] = iso_direct
```

```
                    st["step"] = "agendar_hora"
                    horas = st["horas_por_fecha"].get(iso_direct, [])
                    return f"  {svc._fmt_es(iso_direct)}\ n Elige hora entre: {', '.
    join(horas)}"
            disp = svc.cm.consultar_disponibilidad_proximos_dias(dias=5)
            st["step"] = "agendar_fecha"
            st["opciones_fecha"] = list(disp.keys())
            st["horas_por_fecha"] = disp
            texto = "  Consultando disponibilidad...\n" + "\n".join([f"  {svc.
    _fmt_es(f)}: {', '.join(h) if h else 'Sin huecos'}" for f, h in disp.items()])
            return texto + "\n\ n Elige fecha en **DD/MM/AAAA**."
        if intent == "CONSULTAR_MIS_CITAS":
            citas = svc.cm.listar_citas_paciente(st["dni"], days_ahead=365)
            if not citas:
                disp = svc.cm.consultar_disponibilidad_proximos_dias(dias=5)
                st["step"] = "agendar_fecha"
                st["opciones_fecha"] = list(disp.keys())
                st["horas_por_fecha"] = disp
                texto = "  No tienes citas próximas.\n\n¿Quieres agendar una ahora?
     Responde con la fecha.\n\n"
                texto += "  Consultando disponibilidad...\n" + "\n".join([f"  {svc.
    _fmt_es(f)}: {', '.join(h) if h else 'Sin huecos'}" for f, h in disp.items()])
                return texto
            st["citas"] = citas
            st["step"] = "post_citas"
            return svc._formatear_citas(citas) + "\n\n¿Quieres cancelar (c),
    modificar (m) o salir (s)?"
        if intent == "CANCELAR":
            citas = svc.cm.listar_citas_paciente(st["dni"], days_ahead=365)
            if not citas: return "  No tienes citas próximas."
            st["citas"] = citas; st["step"] = "cancel_select"
            return svc._formatear_citas(citas) + "\n\nEnvía el número de la cita a
    cancelar o 'salir'."
        if intent == "REAGENDAR":
            citas = svc.cm.listar_citas_paciente(st["dni"], days_ahead=365)
            if not citas: return "  No tienes citas próximas."
            st["citas"] = citas; st["step"] = "mod_select"
            return svc._formatear_citas(citas) + "\n\nEnvía el número de la cita a
    reprogramar o 'salir'."
        if intent == "SALUDO":
            st["awaiting_intent"] = True
            return svc._menu_text()
        return self.redirect_message(st, txt, intent)
```

B.5. Validaciones clave (`validators.py`)

Listing B.7: Teléfono ES y DNI/NIE

```
import re
LETRAS_DNI = "TRWAGMYFPDXBNJZSQVHLCKE"
```

```
3
4  def normalizar_telefono_es(t):
5      t = re.sub(r"\D", "", t or "")
6      if t.startswith("34") and len(t) == 11:
7          t = t[2:]
8      if len(t) == 9:
9          return "+34" + t
10     return None
11
12 def validar_telefono_es(t):
13     nt = normalizar_telefono_es(t)
14     if not nt: return None
15     dig = nt[-9:]
16     if not re.match(r"^[6789]\d{8}$", dig): return None
17     return nt
18
19 def validar_dni_nie(doc):
20     if not doc: return None
21     s = doc.strip().upper().replace(" ", "").replace("-", "")
22     if re.match(r"^\d{8}[A-Z]$", s):
23         num = int(s[:8]); letra = s[-1]
24         return s if LETRAS_DNI[num % 23] == letra else None
25     if re.match(r"^[XYZ]\d{7}[A-Z]$", s):
26         mapa = {"X": "0", "Y": "1", "Z": "2"}
27         num = int(mapa[s[0]] + s[1:8]); letra = s[-1]
28         return s if LETRAS_DNI[num % 23] == letra else None
29     return None
```

B.6. Webhook de WhatsApp (`whatsapp_server.py`)

Listing B.8: Extracción de mensajes y envío de respuesta

```
1  import os, time, requests, boto3
2  from typing import Any, Dict, List
3
4  def _extract_messages(payload: Dict[str, Any]) -> List[Dict[str, Any]]:
5      out = []
6      for entry in payload.get("entry", []):
7          for change in entry.get("changes", []):
8              value = change.get("value", {})
9              messages = value.get("messages", []) or []
10             contacts = value.get("contacts", []) or []
11             wa_id = contacts[0].get("wa_id") if contacts else None
12             for m in messages:
13                 msg_from = m.get("from") or wa_id
14                 msg_id = m.get("id")
15                 ts = m.get("timestamp")
16                 mtype = m.get("type")
17                 text = None
18                 if mtype == "text":
```

```
                    text = (m.get("text") or {}).get("body")
                elif mtype == "button":
                    text = (m.get("button") or {}).get("text")
                elif mtype == "interactive":
                    it = m.get("interactive") or {}
                    if "button_reply" in it:
                        text = (it.get("button_reply") or {}).get("title")
                    elif "list_reply" in it:
                        text = (it.get("list_reply") or {}).get("title")
                if msg_from and msg_id and text:
                    out.append({"from": msg_from, "id": msg_id, "timestamp": ts, "
    type": mtype, "text": text})
    return out

def _send_whatsapp_text(to: str, body: str):
    url = f"https://graph.facebook.com/v20.0/{WHATSAPP_PHONE_ID}/messages"
    headers = {"Authorization": f"Bearer {WHATSAPP_TOKEN}", "Content-Type": "
    application/json"}
    data = {"messaging_product": "whatsapp", "to": to, "type": "text", "text": {"
    body": body[:4096]}}
    r = requests.post(url, json=data, headers=headers, timeout=HTTP_TIMEOUT)
    r.raise_for_status()
    return r.json()
```

Listing B.9: Mediación con el servicio del agente

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

app = FastAPI()

@app.post("/webhook")
async def receive(request: Request):
    try:
        payload = await request.json()
    except Exception:
        return JSONResponse(status_code=400, content={"error": "invalid_json"})
    msgs = _extract_messages(payload)
    if not msgs:
        return JSONResponse({"ok": True})
    results = []
    for m in msgs:
        if not _dedupe_accept(m["id"]):
            continue
        agent_payload = {
            "session_id": f"wa:{m['from']}",
            "text": m["text"],
            "metadata": {"channel": "whatsapp", "from": m["from"], "message_id": m["
    id"], "timestamp": m["timestamp"]}
        }
        agent_resp = _call_agent_service(agent_payload)
        replies = []
        if isinstance(agent_resp, dict):
```

```
            if isinstance(agent_resp.get("reply"), str):
                replies = [agent_resp["reply"]]
            elif isinstance(agent_resp.get("replies"), list):
                replies = [str(x) for x in agent_resp["replies"] if x is not None]
        if not replies:
            replies = ["Gracias, estamos procesando tu solicitud."]
        for body in replies:
            try:
                results.append(_send_whatsapp_text(m["from"], body))
            except Exception as e:
                results.append({"error": str(e)})
    return JSONResponse({"ok": True, "results": results})
```

Referencia bibliográfica

[1] World Health Organization. Global diffusion of ehealth: Making universal health coverage achievable, 2016. URL https://apps.who.int/iris/handle/10665/252529.

[2] Statista. Number of monthly active whatsapp users worldwide from 2013 to 2023, 2023. URL https://www.statista.com/statistics/260819/number-of-monthly-active-whatsapp-users/.

[3] Amazon Web Services. Aws lambda developer guide, 2024. URL https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[4] Google. Google calendar api overview, 2023. URL https://developers.google.com/calendar/api.

[5] Cerebras Systems. Cerebras llama 3.3 70b technical overview, 2024. URL https://inference-docs.cerebras.ai/models/llama-33-70b.

[6] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine, 1966. URL https://dl.acm.org/doi/10.1145/365153.365168.

[7] Kenneth Mark Colby. Artificial paranoia: A computer simulation of paranoid processes, 1975. URL https://archive.org/details/artificialparano00colb.

[8] Richard S. Wallace. The anatomy of alice, 2009. URL https://link.springer.com/chapter/10.1007/978-1-4020-6710-5_5.

[9] Asbjørn Følstad and Marita Skjuve. Chatbots for customer service: User experience and motivations, 2019. URL https://dl.acm.org/doi/10.1145/3290607.3313025.

[10] Eleni Adamopoulou and Leonidas Moussiades. An overview of chatbot technology, 2020. URL https://link.springer.com/chapter/10.1007/978-3-030-49186-4_20.

[11] Michael McTear. Conversational ai: Dialogue systems, conversational agents, and chatbots, 2020. URL https://link.springer.com/book/10.1007/978-3-030-39583-4.

[12] Jennifer Zamora. I'm sorry, dave, i'm afraid i can't do that: Chatbot perception and expectations, 2017. URL https://dl.acm.org/doi/10.1145/3027063.3053246.

[13] Dan Jurafsky and James H. Martin. Speech and language processing (draft, 3rd edition), 2023. URL https://web.stanford.edu/~jurafsky/slp3/.

[14] Fernando López, Juan Delgado, and Carmen Pérez. Conversational interfaces for appointment scheduling in health care, 2022. URL https://doi.org/10.1016/j.ijmedinf.2021.104644.

[15] Ioana Baldini, Paul Castro, Kenneth Chang, et al. Serverless computing: Current trends and open problems, 2017. URL https://link.springer.com/chapter/10.1007/978-981-10-5026-8_6.

[16] Eric Jonas, Johannes Schleier-Smith, Vikram Sreekanti, et al. Cloud programming simplified: A berkeley view on serverless computing, 2019. URL https://arxiv.org/abs/1902.03383.

[17] Amazon Web Services. Aws lambda developer guide, 2023. URL https://docs.aws.amazon.com/lambda/.

[18] Matthew Bardsley, Utku Akarca, and Konstantinos Kourtzanidis. Warm: A workflow-aware serverless snapshotting technique, 2021. URL https://ieeexplore.ieee.org/document/9426045.

[19] Mojgan Shahrad, Jialin Balkind, Mingwei Wei, et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, 2020. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

[20] European Union. Reglamento (ue) 2016/679 del parlamento europeo y del consejo, 2016. URL https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[21] Meta Platforms. Whatsapp business platform cloud api documentation, 2022. URL https://developers.facebook.com/docs/whatsapp/cloud-api.

[22] Abhishek Gupta. Building customer support chatbots with whatsapp cloud api, 2023. URL https://www.riverpublishers.com/journal_read_html_article.php?j=JWE/21/4/5.

[23] Meta Platforms. Whatsapp business platform insights, 2024. URL https://developers.facebook.com/docs/whatsapp/insights.

[24] Meta Platforms. Best practices for whatsapp business api security, 2023. URL https://developers.facebook.com/docs/whatsapp/security.

[25] Google LLC. Google calendar api overview, 2023. URL https://developers.google.com/calendar.

[26] Google LLC. Oauth 2.0 policies and best practices, 2022. URL https://developers.google.com/identity/protocols/oauth2.

[27] Nick Michels. Managing time zones in google calendar, 2021. URL https://developers.googleblog.com/2021/07/managing-time-zones-in-google-calendar.html.

[28] Jiang Chen, Tianyu Li, and Xin Wu. Efficient scheduling on google calendar with freebusy query, 2019. URL https://www.researchgate.net/publication/330010747.

[29] Hiroshi Sato, Yutaka Matsuo, and Taro Ishiguro. Idempotent apis in distributed systems for preventing duplicate requests, 2020. URL https://ieeexplore.ieee.org/document/9052374.

[30] Amazon Web Services. Amazon dynamodb developer guide, 2023. URL https://docs.aws.amazon.com/amazondynamodb/.

[31] Amazon Web Services. Aws identity and access management documentation, 2023. URL https://docs.aws.amazon.com/iam/.

[32] Amazon Web Services. Aws secrets manager documentation, 2023. URL https://docs.aws.amazon.com/secretsmanager/.

[33] Amazon Web Services. Aws cloudtrail user guide, 2023. URL https://docs.aws.amazon.com/awscloudtrail/.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

[35] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, et al. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020. URL https://jmlr.org/papers/v21/20-074.html.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

[38] OpenAI. Gpt-4 technical report, 2023. URL https://cdn.openai.com/papers/gpt-4.pdf.

[39] Leo Gao, Zheng Dai, and Jamie Callan. Uncertainty-aware output formats for large language models, 2023. URL https://arxiv.org/abs/2302.06768.

[40] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Program of thoughts eliciting reasoning via large language models, 2023. URL https://arxiv.org/abs/2211.12588.

[41] Susan Zhang, Stephen Roller, Naman Goyal, et al. Opt: Open pre-trained transformer language models, 2022. URL https://arxiv.org/abs/2205.01068.

[42] Kent Beck, Mike Beedle, Arie van Bennekum, et al. Manifesto for agile software development, 2001. URL https://agilemanifesto.org/.

[43] Cloudflare. Cloudflare tunnel quick start, 2023. URL https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/.

[44] Python Software Foundation. Python 3.11 documentation, 2023. URL https://docs.python.org/3/.

[45] Sebastián Ramírez. Fastapi documentation, 2023. URL https://fastapi.tiangolo.com/.

[46] Mangum. Mangum adapter documentation, 2023. URL https://mangum.io/.

[47] Cerebras Systems. Cerebras inference api, 2024. URL https://inference-docs.cerebras.ai/.

[48] Ken Schwaber and Jeff Sutherland. The scrum guide, 2020. URL https://scrumguides.org/scrum-guide.html.

[49] OWASP Foundation. Owasp top ten, 2021. URL https://owasp.org/www-project-top-ten/.

[50] Amazon Web Services. Aws well-architected framework: Security pillar, 2023. URL https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/.

[51] Holger Krekel et al. pytest documentation, 2023. URL https://docs.pytest.org/en/stable/.

[52] David Harel. Statecharts: A visual formalism for complex systems, 1987. URL https://www.sciencedirect.com/science/article/pii/0167642387900359.

[53] Amazon Web Services. Aws lambda pricing, 2025. URL https://aws.amazon.com/lambda/pricing/.

[54] Amazon Web Services. Amazon dynamodb pricing, 2025. URL https://aws.amazon.com/dynamodb/pricing/.

[55] Sebastián Ramírez. Fastapi: Modern, fast web framework for building apis with python, 2025. URL https://fastapi.tiangolo.com/.

[56] Meta Platforms, Inc. Whatsapp cloud api documentation, 2025. URL https://developers.facebook.com/docs/whatsapp/cloud-api.

[57] Cerebras Systems. Cerebras-gpt and llama 3.3 models, 2025. URL https://www.cerebras.net/.