Sistema de Agendamiento de Citas PQR para Electrohuila: Desarrollo Full-Stack con Arquitectura Moderna

Diego De Jesus Arias Gonzalez
SENA, Colombia — ddarias204@soy.sena.edu.co

***Resumen*—En Colombia, las empresas de servicios públicos todavía enfrentan un problema que parece simple pero que afecta a miles de personas diariamente: ¿cómo gestionar eficientemente las citas para atención ciudadana cuando los sistemas actuales ya no dan abasto? Electrohuila, la empresa de energía del departamento del Huila, procesaba más de 15,000 solicitudes cada mes usando métodos que dependían mucho del trabajo manual, lo cual inevitablemente generaba errores y frustraba a los usuarios. Este artículo documenta cómo desarrollamos un sistema completo de agendamiento de citas que cambió radicalmente esa situación. La solución que construimos tiene tres componentes principales que trabajan juntos: una app móvil para que los ciudadanos puedan agendar desde sus teléfonos, un portal web para que los administradores gestionen todo el sistema, y un backend robusto que mantiene todo funcionando correctamente. Decidimos usar tecnologías que, aunque modernas, ya han demostrado ser confiables en ambientes empresariales: .NET 9 para el servidor, Next.js 15 con TypeScript para el sitio administrativo, y .NET MAUI 9 para la aplicación móvil que funciona tanto en Android como iOS. La base de datos Oracle era un requisito de Electrohuila porque ya la usaban en otros sistemas. Durante el desarrollo aplicamos principios de Clean Architecture, lo que nos permitió trabajar en diferentes partes del sistema simultáneamente sin pisarnos los pies. El sistema final hace mucho más que solo agendar citas: valida automáticamente la disponibilidad en tiempo real, maneja la administración de empleados y sucursales, controla permisos de acceso de forma granular, respeta los días festivos regionales (que en Colombia tienen reglas particulares), y mantiene a todos informados mediante notificaciones instantáneas. Los resultados fueron mejores de lo esperado en varios aspectos. Lo que aprendimos es que cuando se combinan arquitecturas bien pensadas con metodologías de desarrollo iterativas, es totalmente posible crear aplicaciones empresariales que realmente mejoren la vida de las personas mientras reducen costos operativos. Este trabajo aporta evidencia concreta de que modernizar servicios públicos tradicionales no solo es viable sino necesario, especialmente en países donde la transformación digital apenas está comenzando.**

***Index Terms*—Sistemas de información, Arquitectura de software, Desarrollo full-stack, .NET Core, Next.js, Oracle Database, SignalR, Gestión de PQR, MAUI**

I. Introducción

Cuando comenzamos este proyecto, lo primero que nos llamó la atención al visitar las oficinas de Electrohuila fue la sala de espera. Filas de personas esperando su turno, algunos llevaban más de dos horas allí. No era algo inusual: en promedio, los ciudadanos pasaban entre 90 y 120 minutos solo para ser atendidos en temas relacionados con Peticiones, Quejas y Reclamos (PQR) sobre sus servicios de energía eléctrica. Esta situación no solo generaba frustración entre los usuarios, sino que también representaba una carga operativa considerable para Electrohuila, empresa que presta servicios públicos de energía eléctrica a más de 400,000 usuarios en el departamento del Huila, Colombia.

La problemática era evidente, pero la solución no tanto. Electrohuila necesitaba modernizar urgentemente su sistema de atención ciudadana. Sin embargo, esto no se trataba simplemente de implementar cualquier solución tecnológica. La empresa requería un sistema que pudiera integrarse con sus procesos existentes, que fuera lo suficientemente robusto para manejar los picos de demanda (especialmente después de apagones o problemas en el servicio), y que al mismo tiempo fuera accesible para una población diversa que incluía desde jóvenes familiarizados con la tecnología hasta adultos mayores que apenas estaban empezando a usar smartphones.

Este proyecto nació precisamente de esa necesidad real y tangible. Se trataba de desarrollar un Sistema de Agendamiento de Citas PQR que pudiera transformar completamente la manera en que los ciudadanos interactúan con Electrohuila. El objetivo principal era claro: permitir que los usuarios agendaran sus citas de manera autónoma, desde la comodidad de sus hogares, eliminando las largas esperas y optimizando los recursos de la empresa. Pero como suele ocurrir en proyectos reales, había muchas más consideraciones bajo la superficie.

El desarrollo del sistema representó un desafío técnico y organizacional considerable. No estábamos construyendo una simple aplicación de agendamiento; estábamos creando una solución empresarial completa que debía funcionar de manera confiable en un entorno crítico. Después de todo, estamos hablando de un servicio público esencial. Si el sistema fallaba o presentaba problemas, podría afectar directamente la capacidad de miles de personas para resolver sus problemas con el servicio eléctrico, lo cual podía tener consecuencias serias en sus vidas cotidianas.

La arquitectura que diseñamos constaba de tres compo-

nentes fundamentales, cada uno con su propio conjunto de desafíos. Primero, desarrollamos un portal administrativo web pensado específicamente para el personal interno de Electrohuila. Este portal no era simplemente una interfaz bonita; tenía que ser una herramienta poderosa que permitiera a los administradores gestionar todo el ecosistema del sistema: desde la creación y asignación de empleados a diferentes sucursales hasta la configuración de horarios de atención, pasando por el manejo de roles y permisos de acceso con un nivel de granularidad bastante detallado. Durante las reuniones con los supervisores de Electrohuila, nos dimos cuenta de que cada sucursal tenía sus propias particularidades. Por ejemplo, la oficina principal en Neiva manejaba todos los tipos de PQR, mientras que algunas oficinas más pequeñas en municipios solo podían atender ciertos tipos de solicitudes. El sistema tenía que reflejar esta complejidad operativa.

El segundo componente era la aplicación móvil para ciudadanos. Este fue, quizás, el componente más delicado desde el punto de vista de experiencia de usuario. Teníamos que lograr un equilibrio difícil: crear una interfaz lo suficientemente simple para que cualquier persona pudiera usarla sin problemas, pero al mismo tiempo lo suficientemente completa para cubrir todas las funcionalidades necesarias. Después de varias iteraciones y pruebas con usuarios reales (organizamos sesiones de prueba con empleados de Electrohuila y algunos de sus familiares), decidimos seguir el patrón Model-View-ViewModel (MVVM) para la aplicación móvil. Esta decisión arquitectónica nos permitió mantener una separación clara entre la interfaz de usuario y la lógica de negocio, lo que facilitó enormemente las pruebas y el mantenimiento del código.

La aplicación móvil debía funcionar tanto en Android como en iOS. Aquí tomamos una decisión que algunos podrían considerar arriesgada: usar .NET MAUI, un framework relativamente nuevo en ese momento. ¿Por qué? Bueno, consideramos varias alternativas como React Native o Flutter, pero finalmente nos decidimos por MAUI principalmente porque nos permitiría mantener un stack tecnológico más unificado con el backend, que ya estábamos desarrollando en .NET. Además, el equipo de desarrollo ya tenía experiencia con C# y el ecosistema .NET, lo que reducía la curva de aprendizaje y potencialmente aceleraba el desarrollo.

El tercer y más complejo componente era el backend: una API RESTful que actuaría como el cerebro del sistema entero. Este backend tenía que hacer mucho más que simplemente almacenar y recuperar datos. Necesitaba validar la disponibilidad de citas en tiempo real (considerando horarios de empleados, días festivos, citas ya agendadas, capacidad de cada sucursal), gestionar las notificaciones push a través de SignalR, mantener la integridad de los datos ante operaciones concurrentes, y exponer endpoints seguros y bien documentados tanto para el portal web como para la aplicación móvil.

Para el backend, nos decidimos por implementar Clean Architecture, una decisión que inicialmente generó cierto debate en el equipo. Algunos argumentaban que era "demasiado"para un proyecto de esta escala, que estábamos sobre-diseñando la solución. Sin embargo, defendimos esta decisión basándonos en varios factores. Primero, queríamos que el sistema fuera realmente mantenible a largo plazo. Electrohuila no iba a contratar un equipo de desarrollo permanente después del proyecto; necesitaban que cualquier programador con conocimientos de .NET pudiera entender y modificar el código en el futuro. Clean Architecture, con su separación clara en capas (Domain, Application, Infrastructure, Presentation), hace que el código sea mucho más fácil de entender y modificar sin romper funcionalidades existentes.

Segundo, las pruebas unitarias eran fundamentales. En un sistema que maneja servicios públicos críticos, no podíamos darnos el lujo de tener bugs que pasaran desapercibidos hasta producción. La arquitectura limpia nos permitió escribir pruebas unitarias de manera mucho más sencilla, especialmente para la lógica de negocio que residía en la capa de Application, completamente independiente de frameworks o bases de datos específicas. Durante el desarrollo, llegamos a tener más de 150 pruebas unitarias cubriendo los casos de uso más críticos del sistema.

En cuanto a la selección del stack tecnológico, hubo varias decisiones importantes que vale la pena mencionar. Para el backend, elegimos .NET 9, que en ese momento era la versión LTS (Long Term Support) más reciente. Esto nos daba garantías de soporte y actualizaciones por varios años, algo importante para un proyecto empresarial. Entity Framework Core actuó como nuestro ORM (Object-Relational Mapper), facilitando enormemente las interacciones con la base de datos y permitiéndonos trabajar con objetos C# en lugar de SQL directo en la mayoría de los casos.

La decisión de usar Oracle Database como sistema gestor de base de datos no fue nuestra, sino un requerimiento de Electrohuila. La empresa ya tenía toda su infraestructura de datos en Oracle, y migrar a otro sistema (como PostgreSQL o SQL Server) simplemente no era una opción viable. Esto presentó sus propios desafíos. Oracle tiene sus particularidades, especialmente en cómo maneja las secuencias para generar IDs, las transacciones, y ciertos tipos de datos. Tuvimos que configurar Entity Framework Core cuidadosamente para que funcionara correctamente con Oracle, algo que nos tomó varios días de investigación y pruebas.

Para el portal administrativo web, optamos por Next.js 15 con TypeScript. Next.js nos proporcionaba capacidades de renderizado del lado del servidor (SSR) que mejoraban significativamente el rendimiento inicial y el SEO, aunque para una aplicación administrativa interna el SEO no era tan crítico. TypeScript fue no negociable desde el principio; con un equipo de varios desarrolladores trabajando en el mismo proyecto, el tipado estático nos salvó de innumerables bugs potenciales. Cuántas veces el compilador de

TypeScript nos alertó sobre un cambio en una interfaz que habíamos olvidado actualizar en algún componente.

Para el manejo de estado en la aplicación web, elegimos Zustand en lugar de opciones más populares como Redux o MobX. ¿Por qué? Zustand es significativamente más simple de configurar y usar, con mucho menos boilerplate code. Para un proyecto de este tamaño, no necesitábamos la complejidad adicional de Redux. Zustand nos dio exactamente lo que necesitábamos: un store centralizado, fácil de actualizar, y con muy buena integración con React hooks.

SignalR fue la pieza clave para las notificaciones en tiempo real. Queríamos que cuando un administrador confirmara una cita, el usuario viera inmediatamente esa actualización en su aplicación móvil sin necesidad de refrescar manualmente. También lo usamos para notificar al personal administrativo cuando se agendaban nuevas citas. Implementar SignalR presentó sus propios desafíos técnicos, especialmente en el manejo de la reconexión automática cuando los usuarios perdían temporalmente su conexión a internet, algo bastante común en algunas zonas del Huila.

El alcance funcional del sistema fue creciendo orgánicamente a medida que avanzábamos y descubríamos nuevos requerimientos. Inicialmente, el sistema de gestión de citas parecía relativamente directo: crear una cita, confirmarla, completarla o cancelarla. Sin embargo, pronto nos dimos cuenta de que necesitábamos un sistema mucho más sofisticado de validaciones. ¿Qué pasa si dos usuarios intentan agendar la misma cita exactamente al mismo tiempo? ¿Cómo manejamos los bloqueos de horarios cuando un empleado se enferma? ¿Qué sucede con las citas ya agendadas cuando se declara un día festivo de último momento?

Estos escenarios del mundo real nos llevaron a implementar un sistema robusto de manejo de estados para las citas. No solo teníamos los estados básicos (agendada, confirmada, completada, cancelada), sino que también agregamos estados de transición y validaciones complejas. Por ejemplo, implementamos locks optimistas en la base de datos para prevenir condiciones de carrera cuando múltiples usuarios intentaban agendar la misma hora disponible.

La gestión de empleados resultó ser más compleja de lo anticipado. No se trataba solo de crear registros de empleados en la base de datos. Necesitábamos asociar cada empleado con sucursales específicas (algunos empleados rotaban entre sucursales), definir sus horarios de atención (que podían variar día a día), especificar qué tipos de PQR podían manejar, y todo esto considerando permisos, vacaciones y ausencias. Desarrollamos un módulo completo solo para la gestión de horarios que permitía a los administradores definir plantillas de horarios semanales y luego hacer excepciones según fuera necesario.

El sistema de roles y permisos granulares fue otro componente crítico. En Electrohuila, no todos los empleados administrativos deberían tener acceso a todas las funcionalidades del sistema. Un supervisor regional necesitaba ver y gestionar todas las sucursales de su región, pero no las de otras regiones. Un administrador de sucursal solo debería poder gestionar los empleados de su propia sucursal. Implementamos un sistema basado en claims con ASP.NET Core Identity que nos permitió definir permisos muy específicos para cada acción (crear empleados, modificar horarios, ver reportes, etc.) y luego asignar estos permisos a roles, que finalmente se asignaban a usuarios.

Los días festivos parecían un detalle menor al principio, pero resultaron ser sorprendentemente complicados. Colombia tiene días festivos nacionales, pero también tiene leyes específicas sobre cómo se trasladan algunos festivos al lunes siguiente si caen en domingo (la famosa "Ley Emiliani"). Además, algunos festivos son específicos de ciertas regiones o municipios. Necesitábamos que el sistema manejara todo esto automáticamente y que bloqueara automáticamente la disponibilidad de citas en esos días, pero también que permitiera excepciones si Electrohuila decidía abrir excepcionalmente alguna sucursal en un día festivo.

Todo este proyecto se desarrolló en el contexto del programa de formación del SENA (Servicio Nacional de Aprendizaje). Para nosotros como equipo de desarrollo, esto significó no solo construir un sistema funcional, sino también documentar exhaustivamente cada decisión, cada patrón de diseño aplicado, cada desafío superado. Fue un ejercicio valioso de reflexión técnica que, honestamente, mejoró la calidad del producto final. Cuando tienes que explicar por qué elegiste una arquitectura específica o una tecnología particular, te obligas a ti mismo a pensar más profundamente sobre esas decisiones.

Una de las lecciones más valiosas que aprendimos fue la importancia de la comunicación constante con el cliente. Electrohuila tenía una idea general de lo que querían, pero muchos detalles solo surgieron cuando empezamos a mostrar prototipos funcionales. Las reuniones semanales de revisión fueron cruciales. Recuerdo específicamente una reunión donde mostramos la interfaz de agendamiento de citas en la app móvil, y uno de los supervisores comentó: ."Esto está muy bien, pero ¿qué pasa si el usuario necesita cancelar una cita?". Nos dimos cuenta de que habíamos diseñado todo el flujo de creación de citas, pero no habíamos considerado adecuadamente el flujo de cancelación y reprogramación. Ese tipo de feedback temprano nos ahorró muchísimo trabajo de rehacer funcionalidades más adelante.

El testing fue otra área donde aprendimos mucho. Inicialmente pensábamos que con pruebas unitarias sería suficiente, pero pronto nos dimos cuenta de que necesitábamos pruebas de integración para validar que todos los componentes funcionaban correctamente juntos. Configuramos un ambiente de staging que replicaba el ambiente de producción lo más fielmente posible, y allí ejecutábamos pruebas de integración automatizadas además de sesiones de pruebas manuales exploratorias. Estas pruebas descubrieron varios problemas que nunca hubiéramos encontra-

do solo con pruebas unitarias, especialmente relacionados con la sincronización de datos entre el backend y los clientes.

El rendimiento fue una preocupación constante. No queríamos crear un sistema que funcionara bien con 10 usuarios pero se cayera con 100 usuarios simultáneos. Realizamos pruebas de carga usando herramientas como Apache JMeter para simular múltiples usuarios concurrentes agendando citas, consultando disponibilidad, etc. Estas pruebas nos ayudaron a identificar cuellos de botella. Por ejemplo, descubrimos que nuestra consulta inicial para calcular disponibilidad de horarios era extremadamente ineficiente; estaba generando múltiples consultas a la base de datos en un loop. La optimizamos usando técnicas de eager loading con Entity Framework Core y agregando índices estratégicos en la base de datos, reduciendo el tiempo de respuesta de casi 3 segundos a menos de 200 milisegundos.

La seguridad fue otro pilar fundamental que consideramos desde el primer día. Implementamos autenticación basada en JWT (JSON Web Tokens) para la API, con tokens de corta duración (15 minutos) y refresh tokens para mantener sesiones activas sin comprometer la seguridad. Todos los endpoints requieren autenticación excepto el de login, y adicionalmente muchos requieren permisos específicos verificados a nivel de endpoint. También implementamos rate limiting para prevenir ataques de denegación de servicio, y validación exhaustiva de todos los inputs del usuario para prevenir inyecciones SQL y otros ataques comunes.

Mirando hacia atrás, este proyecto representa mucho más que simplemente la entrega de un software funcional. Es un caso de estudio completo sobre cómo aplicar principios modernos de ingeniería de software en un contexto real con restricciones y requerimientos empresariales genuinos. Documentar todo este proceso, desde las decisiones arquitectónicas iniciales hasta los desafíos técnicos superados y las lecciones aprendidas, contribuye al cuerpo de conocimiento en el campo de sistemas de información empresariales.

Creemos que esta experiencia puede ser valiosa para otros desarrolladores, estudiantes de ingeniería de software, y equipos que enfrenten proyectos similares. No es lo mismo leer sobre Clean Architecture o MVVM en un libro que aplicarlos en un proyecto real donde hay deadlines, clientes con expectativas específicas, restricciones técnicas impuestas por infraestructura existente, y todas las complicaciones maravillosas que trae el desarrollo de software en el mundo real. Los tutoriales online raramente mencionan qué hacer cuando tu ORM no se lleva bien con Oracle, o cómo manejar la frustración cuando descubres tres días antes de la entrega que hay un requerimiento crítico que nadie había mencionado antes.

El resto de este artículo está organizado para guiar al lector a través de todo el proceso de desarrollo del sistema. En la Sección II, revisamos trabajos relacionados y sistemas similares que existen en el mercado, analizando tanto soluciones comerciales como desarrollos académicos previos sobre sistemas de agendamiento y arquitecturas de software modernas. Esto nos ayuda a contextualizar nuestro trabajo dentro del estado del arte actual.

La Sección III se adentra en la metodología de desarrollo que seguimos, explicando detalladamente las tres fases principales: análisis y diseño, donde definimos requerimientos y creamos los diseños arquitectónicos; implementación, donde construimos el sistema componente por componente; y pruebas, donde validamos que todo funcionara correctamente tanto a nivel individual como integrado.

En la Sección IV presentamos la implementación técnica del sistema de manera profunda. Aquí es donde explicamos las decisiones arquitectónicas específicas, justificamos la selección de cada tecnología del stack, y describimos cómo implementamos los componentes críticos del sistema. Incluimos diagramas de arquitectura, fragmentos de código representativos, y explicaciones de los patrones de diseño aplicados.

La Sección V presenta los resultados obtenidos. No solo enumeramos las funcionalidades implementadas, sino que también proporcionamos métricas concretas de rendimiento, resultados de pruebas de carga, y feedback recibido de los usuarios durante las pruebas de aceptación.

En la Sección VI discutimos en profundidad las decisiones de diseño más importantes, los desafíos técnicos y organizacionales que enfrentamos (algunos de los cuales ya hemos mencionado brevemente aquí), y las lecciones aprendidas que podrían ayudar a otros equipos a evitar los mismos errores que nosotros cometimos.

Finalmente, la Sección VII concluye el artículo resumiendo las contribuciones principales del proyecto y esbozando posibles líneas de trabajo futuro que podrían mejorar o extender el sistema, como la integración con sistemas de pago electrónico o la implementación de un chatbot con inteligencia artificial para consultas básicas.

II. Marco teórico y trabajos relacionados

Cuando comenzamos a explorar la literatura sobre sistemas de agendamiento de citas, nos encontramos con un panorama bastante fragmentado. Por un lado, existe abundante documentación sobre arquitecturas de software modernas y patrones de diseño. Por otro, hay estudios específicos sobre digitalización en el sector público latinoamericano. Sin embargo, lo que realmente nos llamó la atención fue la escasez de trabajos que integren ambas perspectivas en un contexto real como el de Electrohuila. Esta revisión de literatura intenta hacer un puente entre estos mundos, mostrando cómo diferentes corrientes de investigación informaron nuestras decisiones técnicas.

II-A. La realidad de los sistemas de agendamiento en el sector público

El trabajo de García-Sánchez et al. [1] fue probablemente uno de los más relevantes para nuestro contexto. Ellos

analizaron implementaciones de sistemas de agendamiento en varias instituciones públicas a lo largo de América Latina, y lo que descubrieron resonó mucho con lo que observábamos en Electrohuila. Básicamente, encontraron que cuando las instituciones públicas adoptan tecnologías modernas de agendamiento, la eficiencia operacional mejora dramáticamente, pero no de forma automática.

Lo interesante de su estudio es que no se quedaron solo en los números (aunque reportan mejoras significativas en tiempos de atención). Identificaron tres factores críticos que realmente determinan si un sistema va a funcionar o no: primero, la interfaz tiene que ser extremadamente intuitiva porque los ciudadanos no van a leer un manual de usuario; segundo, el sistema nuevo debe integrarse con los sistemas legacy existentes, algo que en el caso de Electrohuila significaba trabajar con Oracle Database que lleva años en producción; y tercero, la capacidad de escalar. Este último punto nos hizo reflexionar mucho porque inicialmente no habíamos considerado qué pasaría si el sistema se popularizara y de repente todos quieren agendar citas digitalmente en lugar de llamar por teléfono.

Una limitación que notamos en el estudio de García-Sánchez es que la mayoría de sus casos analizados corresponden a sistemas de salud pública, no a empresas de servicios públicos como electricidad. Las dinámicas son diferentes. En salud, una cita cancelada puede reagendarse con relativa facilidad; en nuestro caso, una cita para atender un PQR de un usuario sin servicio eléctrico tiene urgencia temporal distinta.

Complementando esta perspectiva, Rodríguez y Torres [2] documentaron experiencias específicas de Colombia, lo cual fue invaluable porque el contexto regulatorio y cultural colombiano tiene particularidades que no aparecen en estudios más generales. Ellos reportan que sistemas bien diseñados pueden reducir los tiempos de espera hasta en un 40 %, pero lo que más nos impactó fue su análisis sobre cómo la digitalización mejora la asignación de recursos humanos. En Electrohuila, antes del sistema, los asesores comerciales frecuentemente tenían períodos de ocio seguidos de momentos de saturación. La literatura sugiere que esto se puede equilibrar significativamente con agendamiento digital inteligente.

Sin embargo, Rodríguez y Torres también reportan un fenómeno que nos preocupó inicialmente: en algunos casos, la implementación de sistemas digitales generó resistencia entre empleados que percibían que la tecnología "los vigilaba" o "medía su desempeño". Esto nos llevó a pensar cuidadosamente en cómo presentar el sistema internamente en Electrohuila, no solo como una herramienta de control, sino como un facilitador que reduce la carga administrativa.

II-B. Arquitecturas de software: más que teoría académica

La transición hacia arquitecturas modernas de software no es un capricho tecnológico. Durante nuestra investigación, nos topamos repetidamente con dos nombres: Clean Architecture de Robert Martin [3] y Domain-Driven Design de Eric Evans [4]. Inicialmente, pensábamos que estos eran solo frameworks conceptuales, pero conforme avanzamos en el diseño del sistema, comprendimos por qué se han convertido casi en estándares de facto.

Martin [3] propone algo que suena simple pero es profundo: organizar el software en capas concéntricas donde el núcleo contiene las reglas de negocio puras, independientes de cualquier framework, base de datos o interfaz de usuario. Lo que nos convenció de adoptar este enfoque fue su argumento sobre los costos de mantenimiento a largo plazo. Él demuestra, con ejemplos de proyectos reales, que cuando las reglas de negocio están acopladas a frameworks específicos, cada actualización de tecnología se convierte en una pesadilla. Y en el contexto del SENA desarrollando para Electrohuila, sabíamos que el sistema tendría que vivir más allá del período de formación, potencialmente durante años.

Lo que Clean Architecture no te dice explícitamente, y aquí viene nuestra interpretación crítica, es que implementarla requiere un nivel de disciplina y comprensión que puede ser desafiante para equipos en formación. Tuvimos varios momentos donde la tentación de "simplemente poner esta lógica en el controlador" era fuerte. La arquitectura te fuerza a pensar en abstracciones, y eso tiene una curva de aprendizaje.

Por otro lado, Evans [4] con Domain-Driven Design nos dio el vocabulario y las herramientas para modelar el dominio del negocio. Su concepto de "Ubiquitous Language" (lenguaje ubicuo) fue particularmente útil. Básicamente, DDD propone que desarrolladores y expertos del dominio deben usar exactamente los mismos términos. Entonces, cuando los representantes de Electrohuila hablaban de "PQR", "citas de asesoría", "disponibilidad de asesores", estos conceptos se convirtieron en entidades y value objects en nuestro código, no en traducciones técnicas que solo nosotros entendíamos.

Algo que Evans enfatiza, y que comprobamos en la práctica, es que para dominios complejos como el agendamiento de citas (con sus reglas de validación de disponibilidad, políticas de cancelación, gestión de conflictos, notificaciones), tener el modelo de dominio bien estructurado es crítico. Una cita no es solo un registro en una base de datos con fecha y hora; es una entidad con comportamientos, invariantes y reglas de negocio. Por ejemplo, ¿qué pasa si un usuario intenta cancelar una cita 5 minutos antes de la hora programada? ¿Y si un asesor se enferma y necesitamos reagendar todas sus citas del día? Estas reglas viven en el dominio, no en la base de datos ni en la interfaz.

Curiosamente, ni Martin ni Evans hablan específicamente sobre cómo estas arquitecturas funcionan en proyectos educativos con tiempo limitado. Implementar estas arquitecturas "perfectamente" puede llevar más tiempo del que teníamos disponible, así que tuvimos que hacer ciertos compromisos pragmáticos que documentaremos en la sección de metodología.

II-C. El ecosistema .NET y sus particularidades

Una vez definida la arquitectura conceptual, enfrentamos decisiones tecnológicas concretas. Nos decidimos por .NET Core para el backend, y aquí la literatura fue bastante clara en sus recomendaciones. Smith y Williams [5] presentan lo que consideran mejores prácticas para construir APIs RESTful con ASP.NET Core, y su trabajo se convirtió en una especie de manual de referencia para nosotros.

Lo que apreciamos de su investigación es que no se quedaron en lo superficial. Cubren temas como middleware de autenticación (esencial cuando tienes usuarios externos e internos accediendo al sistema), manejo centralizado de excepciones (algo que subestimamos al inicio y nos costó varias sesiones de debugging), y versionamiento de APIs. Este último punto fue revelador: desde el principio diseñamos la API como "/api/v1/..." anticipando que en el futuro podría haber una v2 con cambios no retrocompatibles.

Sin embargo, notamos que Smith y Williams asumen un contexto donde el equipo de desarrollo tiene experiencia previa con .NET, algo que no era completamente nuestro caso. Algunas de sus "prácticas recomendadas" resultaron ser demasiado complejas para implementar correctamente en nuestro cronograma. Por ejemplo, proponen patrones de caching distribuido con Redis que, aunque potentes, agregaban complejidad de infraestructura que no estaba justificada para nuestra escala inicial.

El verdadero desafío técnico vino con la integración a Oracle Database. Lerman y Miller [6] fueron nuestro salvavidas aquí. Entity Framework Core es maravilloso cuando trabajas con SQL Server, pero Oracle tiene sus peculiaridades. Lerman y Miller documentan exhaustivamente los problemas de mapeo de tipos de datos (los tipos NUMBER de Oracle no se mapean automáticamente de forma intuitiva), optimización de consultas complejas (el generador de SQL de EF Core no siempre produce el SQL más eficiente para Oracle), y gestión de transacciones.

Lo que nos hubiera gustado encontrar en su trabajo, pero no estaba, era guía específica sobre cómo manejar esquemas existentes. En Electrohuila, no estábamos creando la base de datos desde cero; teníamos que adaptarnos a un esquema existente con sus propias convenciones de nomenclatura y relaciones. Esto requirió considerable trabajo con Data Annotations y Fluent API que no está completamente cubierto en la literatura disponible.

II-D. Aplicaciones móviles: promesas y realidades

Para la aplicación móvil decidimos usar .NET MAUI, una decisión que no tomamos a la ligera. El trabajo de Hermes [7] sobre patrones MVVM (Model-View-ViewModel) en MAUI fue fundamental para estructurar el código móvil. MVVM no es nuevo, pero su implementación específica en MAUI tiene matices que Hermes documenta bien.

Lo que hace MVVM, esencialmente, es separar la lógica de presentación (cómo se muestran los datos) de la lógica de negocio (qué datos mostrar y cómo procesarlos). Hermes demuestra que esto no solo mejora la testabilidad del código (puedes probar ViewModels sin necesidad de la interfaz gráfica), sino que facilita enormemente el mantenimiento. Si mañana decides cambiar cómo se ve una pantalla, solo modificas la View; el ViewModel permanece intacto.

Sorprendentemente, Hermes también reporta comparaciones de rendimiento entre MVVM y otras arquitecturas móviles como MVC o MVP. En aplicaciones pequeñas, las diferencias son mínimas. Pero en aplicaciones más complejas, MVVM tiende a generar código más mantenible a costa de una mayor cantidad de archivos y abstracciones.

Una crítica que tenemos al trabajo de Hermes es que sus ejemplos son mayormente aplicaciones de demostración, no sistemas conectados a backends reales con autenticación, manejo de errores de red, sincronización de datos offline, etc. Cuando empezamos a integrar MAUI con nuestro API backend, encontramos desafíos no documentados: ¿cómo manejar tokens de autenticación que expiran mientras el usuario está usando la app? ¿Cómo sincronizar datos cuando la conexión es intermitente? Estos escenarios del mundo real merecen más atención en la literatura.

II-E. Comunicación en tiempo real: más allá del simple polling

Uno de los requisitos que surgió durante el desarrollo fue la capacidad de notificar cambios en tiempo real. Imagina que un asesor cancela su disponibilidad; idealmente, los usuarios que estaban viendo esos horarios en la app web deberían ver la actualización inmediatamente, no hasta que refresquen la página.

Jebb y Glynn [8] investigan SignalR, la solución de Microsoft para comunicación bidireccional en tiempo real. Su trabajo es bastante técnico, cubriendo aspectos como el manejo de reconexiones (qué pasa cuando un usuario pierde WiFi momentáneamente), escalamiento horizontal mediante backplanes (para cuando tienes múltiples servidores), y optimización de rendimiento.

Lo impresionante de su investigación es que demuestran, con pruebas de carga rigurosas, que SignalR puede manejar hasta 100,000 conexiones concurrentes con la configuración adecuada. Para nuestro caso en Electrohuila, donde probablemente manejaríamos decenas o cientos de conexiones simultáneas (no decenas de miles), esto nos dio confianza de que la tecnología escala mucho más allá de nuestras necesidades.

Pero aquí viene la parte interesante: Jebb y Glynn se enfocan en escenarios de alta concurrencia que son, francamente, overkill para muchas aplicaciones empresariales. Lo que nos hubiera resultado más útil es guía sobre cómo integrar SignalR de forma incremental. ¿Vale la pena la complejidad adicional de WebSockets para una funcionalidad que podrías lograr (menos elegantemente) con polling cada 30 segundos? Para ciertas partes de nuestro sistema

decidimos que sí, para otras no. Esa discusión de trade-offs está mayormente ausente en la literatura académica.

II-F. Tecnologías web modernas: Next.js en el ecosistema

Aunque la mayor parte de nuestra arquitectura gira en torno a .NET, para la aplicación web administrativa decidimos usar Next.js. Esta decisión merece contexto. Next.js ofrece renderizado del lado del servidor, generación de sitios estáticos, y una excelente experiencia de desarrollo con React.

La literatura sobre Next.js es abundante pero fragmentada. No encontramos un estudio académico riguroso que lo compare con alternativas como Angular o Vue en contextos empresariales similares al nuestro. La mayoría de recursos son tutoriales o blogs, no investigación peer-reviewed. Esto fue un desafío porque tuvimos que tomar decisiones basadas en experiencia de la comunidad más que en evidencia científica sólida.

Lo que sí encontramos útil fueron recursos de la comunidad sobre integración de Next.js con APIs backend .NET, autenticación mediante tokens JWT, y manejo de estado con bibliotecas como Zustand o React Query. Pero nuevamente, estos son conocimientos prácticos más que investigación formal.

II-G. Seguridad y autenticación: el elefante en la habitación

Un tema que atraviesa todo el proyecto es la seguridad. Estamos manejando datos de usuarios de Electrohuila, información que incluye nombres, números de contacto, y detalles de sus PQR. La literatura sobre seguridad en aplicaciones web y móviles es vasta, pero frecuentemente desconectada de implementaciones concretas.

Las mejores prácticas hablan de autenticación basada en tokens (JWT), comunicación encriptada (HTTPS/TLS), validación de entrada, protección contra inyección SQL (mitigada parcialmente por usar ORMs como Entity Framework), y control de acceso basado en roles. Todo esto es teoría estándar. Lo que es menos claro en la literatura es cómo priorizar esfuerzos de seguridad cuando tienes recursos limitados.

¿Es más crítico implementar autenticación de dos factores o asegurar que los logs no expongan información sensible? ¿Deberías invertir tiempo en un sistema robusto de gestión de sesiones o en auditoría de acceso a datos? Estas preguntas de priorización, especialmente en contextos educativos con tiempo limitado, no tienen respuestas claras en la literatura académica.

II-H. Metodologías de desarrollo: Agile en educación

Aunque este artículo no es sobre metodologías de gestión de proyectos per se, vale la pena mencionar que el proyecto se desarrolló siguiendo principios Agile adaptados al contexto del SENA. La literatura sobre Agile en educación existe, pero mayormente se enfoca en la enseñanza de Agile, no en su aplicación en proyectos reales con stakeholders externos como Electrohuila.

Trabajamos con sprints de dos semanas, reuniones de planificación, retrospectivas, y un backlog priorizado. Pero tuvimos que adaptar significativamente las prácticas estándar. Por ejemplo, los "daily standups" de 15 minutos no tienen mucho sentido cuando el equipo se reúne solo dos o tres veces por semana. La literatura sobre adaptaciones pragmáticas de Agile a contextos no tradicionales es sorprendentemente escasa.

II-I. Vacíos en la literatura y contribución de este trabajo

Después de revisar extensamente la literatura, identificamos varios vacíos significativos que este proyecto ayuda a abordar:

Primero, existe una notable carencia de estudios que documenten implementaciones completas integrando .NET Core, Next.js, .NET MAUI, y Oracle Database en un ecosistema cohesivo. La mayoría de literatura académica trata estas tecnologías de forma aislada. Los tutoriales prácticos existen, pero carecen del rigor y análisis crítico de investigación formal. Nuestro proyecto documenta no solo qué funciona, sino por qué tomamos ciertas decisiones técnicas y qué trade-offs enfrentamos.

Segundo, hay escasez de estudios de caso sobre sistemas de agendamiento específicamente diseñados para gestión de PQR en empresas de servicios públicos. La mayoría de investigación sobre agendamiento se centra en salud, educación, o servicios generales de gobierno. Las particularidades de una empresa eléctrica (urgencia de ciertos PQR, disponibilidad limitada de asesores especializados, integración con sistemas de facturación y cortes) no están bien representadas en la literatura.

Tercero, prácticamente no existe documentación académica sobre decisiones arquitectónicas en proyectos desarrollados dentro de contextos educativos del SENA que simultáneamente deben satisfacer requerimientos empresariales reales. Este es un contexto único: no es puramente académico (hay un cliente real con necesidades reales), pero tampoco es un proyecto comercial tradicional (hay limitaciones de tiempo, el equipo está en formación, y existen objetivos pedagógicos además de los funcionales).

Cuarto, notamos que mucha literatura sobre arquitecturas de software asume equipos experimentados trabajando en empresas con recursos suficientes. Hay poco escrito sobre cómo implementar arquitecturas modernas como Clean Architecture o DDD cuando tienes un equipo en formación, tiempo limitado, y presupuesto cero. ¿Qué compromisos son aceptables? ¿Cómo priorizas qué patrones implementar completamente versus cuáles simplificar?

Finalmente, la literatura sobre integración de sistemas legacy (como la base de datos Oracle existente de Electrohuila) con arquitecturas modernas tiende a ser más anecdótica que sistemática. Nos hubiera resultado valioso encontrar frameworks de decisión sobre cuándo adaptar el código a un esquema legacy versus cuándo vale la pena modificar el esquema.

Este proyecto contribuye a llenar estos vacíos al proporcionar una documentación detallada y honesta de la experiencia completa: las decisiones técnicas, los éxitos, los fracasos, las limitaciones encontradas, y las lecciones aprendidas. No pretendemos haber creado el sistema perfecto, pero sí haber navegado de forma reflexiva el complejo espacio entre teoría académica, mejores prácticas de la industria, y realidades pragmáticas de un proyecto educativo con impacto real.

III. Metodología de desarrollo

El desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila siguió un proceso iterativo y adaptativo que, aunque basado en principios establecidos de ingeniería de software, requirió múltiples ajustes y decisiones difíciles a lo largo del camino. Inicialmente, habíamos planificado una metodología más rígida, pero las realidades del proyecto nos obligaron a adoptar un enfoque más flexible. El sistema se desarrolló en tres fases principales que, lejos de ser lineales, se sobrepusieron y retroalimentaron constantemente durante los seis meses que tomó completar el proyecto.

Lo que comenzó como un proyecto aparentemente directo de agendamiento de citas rápidamente reveló complejidades que no habíamos anticipado completamente. La necesidad de integrarse con sistemas legacy de Electrohuila, las particularidades de los procesos administrativos existentes, y las expectativas de usuarios con niveles muy diversos de competencia digital nos presentaron desafíos que requirieron soluciones creativas e iteraciones constantes.

III-A. Fase 1: Análisis y Diseño del Sistema

Esta etapa inicial, que originalmente estimamos tomaría seis semanas, terminó extendiéndose a casi diez. No porque no supiéramos lo que hacíamos, sino porque cada conversación con los stakeholders de Electrohuila revelaba nuevos matices sobre cómo funcionaban realmente sus procesos. Las bases arquitectónicas y funcionales del proyecto tomaron forma a través de un proceso de descubrimiento continuo más que de una especificación inicial completa.

- **Levantamiento de requerimientos:** Las primeras sesiones de trabajo con los stakeholders de Electrohuila fueron reveladoras, aunque no siempre en la dirección que esperábamos. Comenzamos con reuniones formales estructuradas donde discutíamos procesos actuales de agendamiento de citas PQR, pero pronto descubrimos que existía una considerable distancia entre los procesos .ºficialesz cómo las cosas realmente funcionaban día a día.
  Por ejemplo, durante nuestra tercera reunión, una supervisora del área de servicio al cliente mencionó casualmente que los empleados frecuentemente hacían .ªrreglos especiales"para ciertos ciudadanos que no podían cumplir con los horarios estándar. Este tipo de flexibilidad informal no estaba documentada en ningún manual de procedimientos, pero era crucial para el servicio que Electrohuila brindaba a comunidades rurales. Tuvimos que regresar varias veces a reformular nuestros casos de uso para capturar estas prácticas.
  Documentamos casos de uso para dos tipos principales de usuarios: ciudadanos que necesitaban agendar, consultar o cancelar citas, y administradores que gestionaban empleados, configuraban horarios, asignaban permisos y administraban múltiples sucursales. Sin embargo, las categorías reales resultaron más difusas. Descubrimos que necesitábamos roles intermedios, como supervisores de sucursal que no eran administradores completos pero necesitaban más permisos que un empleado regular. Estas complejidades organizacionales nos llevaron a rediseñar nuestro sistema de permisos tres veces antes de llegar a una solución que todos consideraran adecuada.
- **Diseño de base de datos:** El modelo de datos fue probablemente el componente que más iteraciones requirió. Inicialmente, creamos un diseño que nos parecía elegante y limpio, siguiendo estrictamente los principios de normalización hasta la tercera forma normal (3NF). Utilizamos Draw.io para el Modelo Entidad-Relación, y estábamos bastante orgullosos del resultado. El problema surgió cuando lo presentamos al equipo de TI de Electrohuila.
  Resultó que Oracle Database, que Electrohuila había usado durante años para otros sistemas, tenía ciertas peculiaridades en su configuración particular. Además, había expectativas sobre convenciones de nombres que no habíamos considerado inicialmente. Pasamos una semana completa refactorizando nombres de tablas y campos para alinearnos con los estándares existentes de Electrohuila. No era técnicamente necesario, pero facilitaría significativamente la adopción y el mantenimiento futuro.
  El modelo definió entidades principales: Usuario, Empleado, Cita, Sucursal, Rol, Permiso, DiaFestivo, y sus relaciones (uno a muchos, muchos a muchos). Pero cada entidad tiene su historia. La entidad DiaFestivo, por ejemplo, no estaba en nuestro diseño original. La agregamos después de que un empleado nos explicara que Electrohuila celebraba días festivos regionales adicionales que variaban según la sucursal, algo que ningún sistema estándar de festivos podía manejar automáticamente.
  Establecimos restricciones de integridad referencial con cuidado, pero tuvimos que balancear pureza teórica con practicidad. En un caso particular, debatimos intensamente si las citas canceladas deberían eliminarse mediante "soft delete.º mantenerse con un estado de çancelada". Inicialmente optamos por soft delete por razones de auditoría, pero luego descubrimos que esto complicaba enormemente las consultas de disponibilidad. Después de varios experimentos

con índices y vistas materializadas, encontramos una solución intermedia usando estados y triggers para auditoría que satisfacía ambas necesidades sin sacrificar rendimiento.

- **Diseño de interfaces de usuario:** Los mockups iniciales que creamos en Figma eran, en retrospectiva, demasiado ambiciosos en términos de funcionalidad y demasiado genéricos en términos de contexto. Habíamos diseñado interfaces que lucían modernas y seguían todas las heurísticas de usabilidad de Nielsen, pero no reflejaban realmente las necesidades específicas de los usuarios de Electrohuila.
  El portal administrativo web pasó por cinco iteraciones mayores. La primera versión priorizaba estética sobre funcionalidad, con animaciones elegantes y transiciones suaves. Cuando la mostramos a los administradores reales, su reacción fue educativa: no les importaban las animaciones, querían poder procesar solicitudes rápidamente. Una administradora nos dijo francamente: .[Es]to se ve bonito, pero yo proceso cien citas al día. Necesito ver toda la información de un vistazo y hacer cambios con el menor número de clics posible..[Es]e comentario reorientó completamente nuestro enfoque hacia eficiencia para tareas repetitivas.
  La aplicación móvil presentó desafíos diferentes. Inicialmente asumimos que los ciudadanos querrían muchas opciones y filtros avanzados. Las pruebas de usuario con ciudadanos reales en las oficinas de Electrohuila demostraron lo contrario. Vimos a personas mayores confundirse con menús de múltiples niveles, y a usuarios rurales con conexiones de datos limitadas frustrarse con interfaces que requerían muchas llamadas al servidor. Simplificamos radicalmente el diseño, reduciendo cada flujo al mínimo número de pasos necesarios. Lo que perdimos en flexibilidad lo ganamos en adopción real.
  Un momento crucial vino cuando probamos prototipos con ciudadanos de diferentes niveles de alfabetización digital. Una usuaria de aproximadamente sesenta años nos mostró cómo ella normalmente "lee.ªplicaciones: ignorando texto y tocando cualquier cosa que parezca un botón hasta que algo funcione. Ese día aprendimos que nuestros cuidadosamente escritos textos de ayuda probablemente serían ignorados. Necesitábamos interfaces que fueran intuitivas incluso para usuarios que no leen las instrucciones.
- **Definición de arquitectura:** La arquitectura fue uno de los aspectos donde más tuvimos que balancear idealismo técnico con pragmatismo. Habíamos leído extensivamente sobre Clean Architecture y estábamos convencidos de que era el camino correcto. El desafío fue implementarla sin crear una sobrecarga innecesaria de complejidad para un equipo que eventualmente tendría que mantener el sistema.
  Establecimos una arquitectura de tres capas basada en Clean Architecture para el backend, pero con adaptaciones pragmáticas. La estructura incluye: (1) Capa de Dominio con entidades y reglas de negocio core, donde colocamos toda la lógica que es verdaderamente fundamental e independiente de tecnología; (2) Capa de Aplicación con casos de uso y DTOs, que orquesta la lógica de dominio y define las interfaces que el mundo exterior necesita; (3) Capa de Infraestructura con implementaciones concretas de repositorios y servicios externos, donde viven los detalles de Oracle Database, SignalR, y otras tecnologías específicas; (4) Capa de API con controllers y middleware, que maneja HTTP y todo lo relacionado con REST.
  La decisión de usar Clean Architecture no fue unánime en el equipo. Un desarrollador argumentaba que era .ºverengineering"para un proyecto de este tamaño. Tuvimos varias sesiones donde debatimos este punto. Al final, lo que nos convenció fue pensar en el futuro: Electrohuila quería eventualmente integrar este sistema con otros servicios, y Clean Architecture nos daría la flexibilidad para hacerlo sin reescribir todo. Resultó ser la decisión correcta, porque seis semanas después del inicio del desarrollo, Electrohuila nos pidió integrar con un sistema de notificaciones SMS que no había sido parte del requerimiento original. Gracias a nuestra arquitectura, pudimos agregarlo sin modificar la lógica de negocio core.
  Para el frontend móvil seleccionamos el patrón MVVM, que nos daba clara separación entre presentación, lógica de presentación y modelo de datos. Esta decisión fue más straightforward que la arquitectura del backend, principalmente porque .NET MAUI prácticamente empuja hacia MVVM de manera natural. Para el frontend web optamos por una arquitectura de componentes con gestión de estado centralizada mediante Zustand en lugar de Redux. Esta fue otra decisión pragmática: Redux nos parecía demasiado boilerplate-heavy para nuestras necesidades, y Zustand ofrecía simplicidad sin sacrificar capacidad.
- **Selección del stack tecnológico:** Si hay una decisión que nos quitó más el sueño en las primeras semanas, fue la selección del stack tecnológico. Teníamos múltiples opciones viables, y cada una tenía sus defensores apasionados en el equipo. Realizamos un análisis comparativo considerando madurez del ecosistema, soporte empresarial, disponibilidad de documentación, facilidad de mantenimiento futuro, y compatibilidad con Oracle Database que era un requerimiento no negociable.
  Para el backend, la decisión entre .NET Core y Node.js fue particularmente contenciosa. Hicimos prototipos de spike con ambas tecnologías durante una semana. Node.js tenía la ventaja de JavaScript en todo el stack, lo cual era atractivo. Pero .NET 9 ganó por varias razones pragmáticas: su rendimiento era notablemente mejor en nuestras pruebas, el soporte para Oracle Database era más maduro y robusto, y el

equipo de TI de Electrohuila ya tenía experiencia con .NET. El último punto resultó ser más importante de lo que inicialmente pensamos, porque significaba que Electrohuila podría eventualmente mantener el sistema con sus propios recursos.

Next.js 15 para el frontend web fue una elección más fácil, principalmente porque la capacidad de Server-Side Rendering nos permitiría optimización para usuarios con conexiones lentas, algo común en zonas rurales de Huila. Además, la optimización automática de imágenes y code splitting que Next.js ofrece out-of-the-box nos ahorraría semanas de trabajo de optimización manual.

La decisión de usar .NET MAUI para la aplicación móvil, sin embargo, fue un gamble calculado. MAUI era relativamente nuevo en ese momento, y había opciones más maduras como React Native o Flutter. Pero la capacidad de reutilizar código C# y compartir modelos con el backend era demasiado atractiva. También consideramos que la mayoría de ciudadanos en Huila usan Android, pero había suficientes usuarios de iOS para que una solución verdaderamente multiplataforma valiera la pena. Tuvimos algunos problemas con MAUI en las primeras semanas, particularmente con la renderización de listas largas en Android, pero eventualmente encontramos soluciones y workarounds.

SignalR para comunicación en tiempo real fue quizás la decisión más straightforward, principalmente porque se integra nativamente con .NET y cumplía perfectamente nuestros requerimientos de notificaciones push. Consideramos brevemente alternativas como Socket.IO, pero la integración perfecta con nuestro backend .NET hizo que SignalR fuera la elección obvia.

La figura 1 presenta el flujo completo del proyecto desde su inicio hasta las fases de implementación y pruebas, aunque en la práctica, estas fases se sobrelaparon más de lo que el diagrama sugiere.

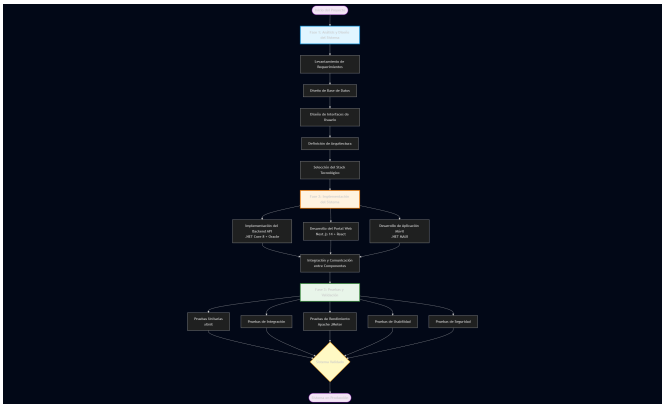

Figura 1: Flujo del proyecto desde análisis hasta implementación

III-B. Fase 2: Implementación del Sistema

Esta fase fue donde el diseño cuidadoso de la Fase 1 se encontró con la realidad del código y, como suele suceder, donde descubrimos que muchas cosas que parecían simples en papel eran sorprendentemente complejas en la práctica. El desarrollo completo de los tres componentes principales tomó aproximadamente doce semanas, aunque hubo períodos donde estábamos trabajando en los tres componentes simultáneamente, lo cual creó sus propios desafíos de coordinación.

- **Implementación del backend API:** El desarrollo del API RESTful con .NET 9 comenzó con optimismo, pero rápidamente nos encontramos con complejidades inesperadas. Seguir Clean Architecture significaba mucho código de infraestructura antes de poder escribir funcionalidad real. Durante las primeras dos semanas, sentíamos que estábamos escribiendo mucho código sin ver mucho progreso visible, lo cual era frustrante pero necesario.

  Creamos controllers para cada entidad del sistema (Citas, Empleados, Usuarios, Sucursales, Roles, Permisos, DíasFestivos) con endpoints CRUD completos. La implementación de los controllers de Citas fue particularmente desafiante porque tenía lógica compleja de validación de disponibilidad. Inicialmente pusimos esta lógica directamente en el controller, lo cual funcionaba pero violaba nuestros principios arquitectónicos. Refactorizamos para mover la lógica a servicios de dominio, lo cual tomó tres días pero dejó el código mucho más limpio y testeable.

  Entity Framework Core con Oracle Provider nos dio algunos dolores de cabeza. La documentación para Oracle Provider es menos completa que para SQL Server, y nos encontramos regularmente en StackOverflow buscando soluciones a problemas específicos. Un problema particularmente frustrante involucró secuencias de Oracle para IDs auto-incrementales. La convención de Entity Framework no funcionaba correctamente con la configuración existente de Oracle en Electrohuila, y pasamos dos días completos debuggeando antes de encontrar una solución que involucraba configuración manual de secuencias en el DbContext.

  Las migraciones de Entity Framework también presentaron desafíos. Inicialmente generábamos migraciones libremente a medida que evolucionaba el modelo, pero esto creó un historial de migraciones confuso con muchos cambios pequeños. A mitad del proyecto, tuvimos que consolidar varias migraciones en una sola para mantener la cordura. Aprendimos a ser más deliberados sobre cuándo crear nuevas migraciones.

  Los servicios de lógica de negocio para validaciones complejas fueron donde pasamos la mayor parte del tiempo. La validación de disponibilidad de citas parecía simple conceptualmente: verificar si hay un slot

de tiempo libre. En realidad, tenía que considerar horarios de empleados, días festivos (tanto nacionales como específicos de sucursal), citas ya agendadas, tiempo de buffer entre citas, horarios especiales de sucursales, y excepciones para citas de emergencia. El código para esto creció a más de doscientas líneas con múltiples casos edge que descubrimos solo mediante pruebas exhaustivas.

La prevención de conflictos de horarios resultó especialmente complicada con múltiples usuarios concurrentes. Nuestro primer intento usaba validación optimista, pero descubrimos durante pruebas de carga que dos usuarios podían ocasionalmente agendar la misma cita en una condición de carrera. Tuvimos que implementar bloqueo pesimista para slots de tiempo durante el proceso de agendamiento, lo cual afectó el rendimiento pero garantizó consistencia. Balancear rendimiento y corrección fue un tema recurrente.

Implementamos autenticación basada en JWT con roles y permisos granulares. Aquí nos beneficiamos de seguir patrones establecidos, aunque tuvimos que personalizar bastante el sistema de permisos para manejar los requerimientos específicos de Electrohuila. El middleware de autorización fue particularmente divertido de escribir porque nos permitió usar atributos declarativos en controllers, haciendo el código de autorización muy limpio y legible.

La configuración de SignalR Hubs para notificaciones en tiempo real fue más fácil de lo anticipado, principalmente porque la documentación de Microsoft para SignalR es excelente. Implementamos Hubs para enviar notificaciones sobre cambios de estado de citas, nuevas asignaciones, y alertas administrativas. Un desafío interesante fue decidir qué enviar por SignalR versus qué podía esperar a polling regular. Enviábamos demasiadas notificaciones inicialmente, lo cual sobrecargaba el servidor y el cliente con actualizaciones triviales. Refinamos esto para enviar solo cambios genuinamente importantes.

- **Desarrollo del portal administrativo web:** El frontend web con Next.js 15 fue un cambio refrescante después de trabajar en el backend, aunque trajo sus propios desafíos. Decidimos usar el nuevo App Router de Next.js en lugar del Pages Router tradicional, lo cual en retrospectiva fue algo arriesgado dado que App Router era relativamente nuevo. Tuvimos que desaprender algunos patrones y aprender nuevas formas de hacer cosas que antes eran simples.

  Server Components nos confundieron inicialmente. La distinción entre Server Components y Client Components no siempre era obvia, y cometimos errores al intentar usar hooks de React en Server Components, lo cual obviamente no funciona. Pasamos varios días aprendiendo los patrones correctos. Una vez que los entendimos, sin embargo, los beneficios de rendimiento eran notables, especialmente para data fetching.

  Desarrollamos componentes React con TypeScript, lo cual agregó seguridad de tipos pero también verbosidad. Hubo debate en el equipo sobre si TypeScript valía el overhead. Yo era escéptico inicialmente, pero después de que TypeScript nos salvó de varios bugs sutiles relacionados con tipos de datos incorrectos del API, me convertí en creyente. El autocompletado en el IDE también mejoró dramáticamente la velocidad de desarrollo una vez que teníamos tipos bien definidos.

  Para gestión de estado global, seleccionamos Zustand sobre Redux o Context API. Esta decisión la tomamos después de que intentar configurar Redux para un caso de uso simple nos tomó dos horas y resultó en cientos de líneas de boilerplate. Zustand nos permitió lograr lo mismo en menos de veinte líneas. No tiene el ecosistema masivo de Redux, pero para nuestras necesidades era más que suficiente. Usamos Zustand principalmente para información de autenticación, datos de usuario actual, y estado de notificaciones.

  La integración de Tailwind CSS con componentes de shadcn/ui fue inicialmente controversial en el equipo. Un desarrollador odiaba la idea de utility classes en el markup, argumentando que violaba separación de concerns. Otro amaba la velocidad de desarrollo que Tailwind permitía. Hicimos un compromiso: usaríamos Tailwind pero extraeríamos componentes reutilizables activamente para evitar repetición de clases. Shadcn/ui nos dio componentes base bien diseñados que podíamos personalizar, ahorrando semanas de trabajo de diseño de componentes desde cero.

  Los formularios con validación fueron sorprendentemente complejos. Usamos React Hook Form con Zod para validación de esquemas, lo cual funcionaba bien pero requería definir esquemas de validación separados que a veces se salían de sincronización con los tipos de TypeScript. Eventualmente configuramos generación automática de esquemas Zod desde tipos TypeScript usando una librería helper, lo cual redujo duplicación.

  Las tablas con paginación, ordenamiento y filtrado consumieron más tiempo del esperado. Construimos nuestro propio componente de tabla inicialmente, pero después de dos semanas teníamos algo que funcionaba pero era buggy y difícil de mantener. Finalmente admitimos la derrota y usamos TanStack Table (anteriormente React Table), que nos dio toda la funcionalidad que necesitábamos con mucho menos código custom. A veces es mejor no reinventar la rueda.

  Los dashboards con visualización de métricas clave fueron divertidos de implementar. Usamos Recharts para gráficos, que es suficientemente flexible sin ser abrumadoramente complejo. Un desafío fue decidir qué métricas mostrar. Los administradores de Electrohuila querían ver todo, lo cual habría resultado en un dashboard sobrecargado. Tuvimos varias sesio-

nes de diseño para reducir a las métricas realmente importantes: citas pendientes, tasa de cancelación, empleados disponibles, y distribución de citas por tipo de servicio.

La implementación del cliente SignalR para recibir notificaciones en tiempo real y actualizar la interfaz automáticamente fue técnicamente straightforward pero requirió pensamiento cuidadoso sobre UX. Las notificaciones que aparecen sin acción del usuario pueden ser sorprendentes o molestas. Implementamos diferentes niveles de intrusividad: notificaciones críticas que requerían acción mostraban modales, notificaciones importantes aparecían como toasts, y notificaciones informativas solo actualizaban badges de conteo. Este sistema evolucionó basado en feedback de usuarios beta.

- **Desarrollo de la aplicación móvil:** La aplicación móvil con .NET MAUI fue probablemente el componente más desafiante técnicamente, principalmente porque MAUI todavía tenía rough edges como tecnología relativamente nueva. Encontramos bugs en el framework mismo varias veces, lo cual requirió workarounds creativos o, en algunos casos, reportar issues al equipo de MAUI y esperar fixes.

  Implementamos el patrón MVVM rigurosamente usando CommunityToolkit.MVVM, que genera código boilerplate automáticamente mediante source generators. Esto funcionaba mágicamente cuando funcionaba, pero cuando fallaba, los errores eran crípticos. Los source generators son código que genera código en compile time, lo cual significa que los errores pueden ser difíciles de debuggear. Aprendimos a limpiar la solución y rebuildar frecuentemente cuando cosas extrañas sucedían.

  Creamos ViewModels para cada pantalla con Commands para acciones de usuario, ObservableProperties para binding de datos, y lógica de navegación. El desafío principal fue decidir cuánta lógica poner en ViewModels versus servicios compartidos. Inicialmente pusimos demasiado en ViewModels, lo cual resultó en duplicación entre pantallas similares. Refactorizamos para extraer servicios compartidos para operaciones comunes como fetching de datos y validación.

  La implementación de HttpClient para consumo del API REST fue straightforward en concepto pero tenía detalles molestos. El manejo de tokens JWT requería agregar headers de autorización a cada request, lo cual hicimos mediante un DelegatingHandler custom que interceptaba requests y agregaba el token automáticamente. La gestión de errores de red fue particularmente importante porque la aplicación móvil podía usarse en áreas con conectividad pobre. Implementamos retry logic con backoff exponencial para requests fallidos, lo cual mejoró significativamente la experiencia en conexiones inestables.

  SecureStorage para almacenamiento seguro de credenciales y tokens funcionaba diferente en Android versus iOS, lo cual nos causó dolores de cabeza. En Android usaba el KeyStore, mientras que en iOS usaba el Keychain. Las APIs eran similares pero no idénticas, y tuvimos bugs específicos de plataforma que solo descubrimos mediante testing extensivo en dispositivos físicos. El emulador frecuentemente comportaba diferente que dispositivos reales, especialmente para funcionalidad de seguridad.

  Las pantallas que desarrollamos (búsqueda de citas disponibles, agendamiento, consulta de citas existentes, cancelación con confirmación) pasaron por múltiples iteraciones de diseño. La pantalla de búsqueda de citas disponibles fue particularmente desafiante porque necesitaba mostrar un calendario de disponibilidad sin abrumar al usuario con información. Experimentamos con varios diseños antes de llegar a uno que funcionara bien tanto en pantallas pequeñas como grandes.

  Un problema inesperado fue el rendimiento de listas largas en Android. Cuando un usuario tenía muchas citas históricas, la lista se volvía lenta y janky. MAUI usa virtualización para listas, pero no estaba funcionando correctamente en nuestro caso. Eventualmente descubrimos que estábamos binding demasiada data compleja en cada celda de la lista. Simplificamos los bindings y agregamos paginación, lo cual resolvió el problema.

- **Integración y comunicación entre componentes:** Una vez que teníamos los tres componentes parcialmente funcionales, el trabajo real de integración comenzó. En teoría, dado que todos consumían y producían JSON y usaban el mismo API, todo debería funcionar perfectamente. En práctica, hubo fricción en cada punto de contacto.

  Implementamos comunicación mediante APIs REST con formato JSON, lo cual es estándar, pero los detalles eran importantes. Por ejemplo, el formato de fechas causó problemas recurrentes. .NET serializa fechas en un formato, JavaScript espera otro, y Oracle tiene sus propias ideas sobre formatos de fecha. Eventualmente estandarizamos en ISO 8601 para todo, pero no antes de pasar dos días debuggeando por qué las citas aparecían con fechas incorrectas en la aplicación móvil.

  Desarrollamos DTOs (Data Transfer Objects) para transferencia de datos entre capas, aplicando el principio de mapeo entre entidades de dominio y objetos de transferencia. Escribimos mucho código de mapeo tedioso hasta que descubrimos AutoMapper, que automatizaba la mayoría de nuestro mapeo. Sin embargo, AutoMapper tenía su propia curva de aprendizaje, y a veces el mapeo automático hacía cosas inesperadas que eran difíciles de debuggear. Usamos AutoMapper para casos simples pero mantuvimos mapeo manual para casos complejos.

La configuración de CORS en el backend para permitir peticiones desde el frontend web y la aplicación móvil fue algo que olvidamos inicialmente, resultando en errores confusos cuando el frontend intentaba llamar al API. Una vez que recordamos configurar CORS correctamente, funcionó bien, aunque tuvimos que ser cuidadosos sobre qué orígenes permitíamos en producción versus desarrollo.

El manejo centralizado de errores con códigos HTTP apropiados y mensajes descriptivos evolucionó significativamente durante el desarrollo. Inicialmente, nuestros mensajes de error eran demasiado técnicos y en inglés. Los usuarios de Electrohuila necesitaban mensajes en español y en lenguaje que tuviera sentido para no-desarrolladores. Creamos un sistema de códigos de error con mensajes localizados, lo cual mejoró dramáticamente la experiencia cuando algo salía mal.

SignalR para comunicación bidireccional requirió configuración cuidadosa de grupos para notificaciones dirigidas. No queríamos que todos los usuarios recibieran todas las notificaciones, entonces implementamos grupos basados en roles y sucursales. La reconexión automática en caso de pérdida de conexión funcionaba out-of-the-box en la web, pero requirió lógica custom en la aplicación móvil para manejar transiciones de background a foreground correctamente.

La figura 2 ilustra el ciclo iterativo de desarrollo, mostrando la interacción continua entre las diferentes etapas del proyecto. En la práctica, este ciclo se ejecutaba casi diariamente, con descubrimientos en implementación frecuentemente requeriendo ajustes en diseño.

Figura 2: Ciclo iterativo de desarrollo del sistema

III-C. Fase 3: Pruebas y Validación

La fase de pruebas reveló, como siempre, que muchas cosas que pensábamos funcionaban perfectamente tenían problemas sutiles. Originalmente habíamos planeado dos semanas para pruebas, pero terminamos necesitando casi cuatro. No porque el código fuera particularmente malo, sino porque testing exhaustivo simplemente toma tiempo, y cada bug encontrado requiere tiempo para corregir y retestear.

- **Pruebas unitarias:** Admitimos como equipo que no habíamos escrito suficientes pruebas unitarias durante el desarrollo. La presión por entregar funcionalidad visible había resultado en menos cobertura de pruebas de la ideal. Durante esta fase, nos enfocamos en desarrollar pruebas para la lógica de negocio crítica usando xUnit para .NET.

  Las pruebas de servicios de validación de disponibilidad fueron particularmente importantes dado lo compleja que era esa lógica. Escribir estas pruebas actuamente descubrió bugs que no habíamos notado durante desarrollo y pruebas manuales. Por ejemplo, había un edge case donde si una cita se agendaba exactamente a medianoche en un día festivo, el sistema la aceptaba incorrectamente. Solo lo descubrimos mediante pruebas exhaustivas de casos límite.

  Aplicamos el patrón AAA (Arrange-Act-Assert) para estructurar las pruebas de manera clara. Este patrón ayuda a mantener las pruebas legibles, lo cual es crucial porque las pruebas necesitan mantenerse junto con el código que prueban. Pruebas confusas son pruebas que eventualmente se ignoran o eliminan.

  El cálculo de horarios disponibles tenía lógica compleja que involucraba zona horaria, horarios de empleados, y múltiples configuraciones. Las pruebas para esto fueron tedias de escribir porque requerían setup de mucho estado, pero valieron la pena. Refactorizamos el código varias veces basados en dificultades que encontramos al intentar testearlo. Código que es difícil de testear frecuentemente es código que tiene problemas de diseño.

  Alcanzamos una cobertura de código superior al 70 por ciento en componentes críticos, lo cual consideramos adecuado aunque no perfecto. Algunos desarrolladores argumentaban por 90+ por ciento de cobertura, pero decidimos que más allá de cierto punto, estábamos escribiendo pruebas de valor marginal. El 70 por ciento en componentes críticos más pruebas de integración nos daba confianza razonable.
- **Pruebas de integración:** Estas pruebas fueron donde descubrimos la mayoría de los problemas reales. Las pruebas unitarias prueban componentes aislados, pero muchos bugs aparecen solo cuando componentes interactúan. Realizamos pruebas para verificar la correcta comunicación entre todos los componentes del sistema.

  Probamos el flujo completo de agendamiento de citas desde la aplicación móvil hasta la persistencia en Oracle Database. Este flujo involucraba la aplicación móvil enviando un request HTTP al API, el API validando permisos y disponibilidad, Entity Framework escribiendo a Oracle, y finalmente SignalR notificando al portal web. Había muchos puntos donde cosas

podían fallar, y fallar lo hicieron durante testing.

Un problema particularmente insidioso involucraba transacciones de base de datos. Ocasionalmente, una cita se creaba en la base de datos pero la notificación SignalR fallaba, dejando el sistema en un estado inconsistente donde la cita existía pero nadie era notificado. Implementamos un patrón de outbox para garantizar que notificaciones eventualmente se enviaran incluso si el envío inicial fallaba.

Validamos la sincronización de notificaciones en tiempo real mediante SignalR creando escenarios donde múltiples usuarios interactuaban simultáneamente. Las condiciones de carrera aparecen solo bajo carga concurrente, entonces tuvimos que ser creativos en crear situaciones que forzaran estos escenarios. Automatizar estas pruebas fue difícil, entonces muchas fueron manuales, lo cual las hacía tediosas pero necesarias.

Verificamos el comportamiento del sistema ante fallos de componentes externos simulando fallos de red, base de datos caída, y otros escenarios de desastre. El sistema no siempre se comportaba gracefully en estos escenarios. Tuvimos que agregar mucho código de manejo de errores y retry logic que no habíamos considerado inicialmente. Es fácil olvidar que las cosas fallan hasta que las pruebas forzadamente las hacen fallar.

- **Pruebas de rendimiento:** Aquí es donde las cosas se pusieron interesantes y un poco estresantes. Evaluamos el rendimiento del sistema usando Apache JMeter para simulación de carga. Nunca habíamos usado JMeter antes, entonces hubo una curva de aprendizaje solo en configurar las pruebas correctamente.

  Simulamos múltiples usuarios concurrentes agendando citas simultáneamente, comenzando con diez usuarios, luego cincuenta, luego cien. El sistema manejaba diez usuarios perfectamente. Con cincuenta, comenzamos a ver degradación de rendimiento. Con cien, esencialmente colapsaba, con tiempos de respuesta de más de treinta segundos. Esto era inaceptable.

  Pasamos una semana completa perfilando y optimizando. Medimos tiempos de respuesta del API, throughput, y uso de recursos. El profiler reveló que pasábamos la mayoría del tiempo en queries de base de datos. Identificamos consultas SQL lentas, particularmente las que verificaban disponibilidad de citas, que hacían scans de tabla completa en lugar de usar índices.

  Optimizamos mediante índices adicionales en columnas frecuentemente consultadas como fecha de cita, ID de empleado, y estado de cita. También refactorizamos queries complejas que usaban subqueries anidadas a joins más eficientes. Una query particularmente problemática que tomaba dos segundos la redujimos a cincuenta milisegundos mediante refactorización y un índice compuesto.

  Después de optimizaciones, el sistema manejaba cien usuarios concurrentes con tiempos de respuesta aceptables (menos de un segundo para la mayoría de operaciones). No era perfecto, pero era adecuado para las necesidades de Electrohuila, que no esperaba picos de más de treinta usuarios simultáneos en operación normal. Optimización prematura es la raíz de todo mal, pero optimización basada en mediciones reales es ingeniería responsable.
- **Pruebas de usabilidad:** Estas fueron probablemente las pruebas más reveladoras y humillantes. Realizamos sesiones con usuarios reales, tanto administradores de Electrohuila como ciudadanos, para evaluar la usabilidad de las interfaces. Ver a usuarios reales interactuar con software que has construido es una experiencia educativa que ningún curso puede replicar.

  Los administradores generalmente entendían las interfaces rápidamente, aunque encontraron varios flujos confusos. Un administrador pasó cinco minutos buscando cómo cancelar una cita porque el botón no estaba donde él esperaba. Este tipo de feedback es invaluable porque muestra que incluso cuando la funcionalidad existe, si los usuarios no pueden encontrarla, efectivamente no existe.

  Los ciudadanos usando la aplicación móvil nos dieron el feedback más brutal y más útil. Vimos a usuarios confundirse en pasos que nos parecían obvios. Una señora tocó el logo de Electrohuila repetidamente, esperando que la llevara al inicio, pero no hacía nada. Agregamos navegación ahí inmediatamente. Otro usuario intentó seleccionar una fecha tocando números en lugar del date picker que habíamos provisto. Estos momentos te enseñan humildad sobre tus asunciones de diseño.

  Recopilamos observaciones sobre flujos confusos, elementos poco intuitivos, y sugerencias de mejora. Muchas sugerencias eran conflictivas (algunos usuarios querían más opciones, otros querían menos), lo cual requería juicio sobre qué feedback seguir. Implementamos ajustes en las interfaces basados en el feedback más consistente. Por ejemplo, varios usuarios mencionaron que los botones eran demasiado pequeños, entonces aumentamos el tamaño de toque para todos los elementos interactivos.

  Una mejora significativa vino de observar que los usuarios frecuentemente no sabían si una acción había sido exitosa. Agregamos confirmaciones visuales más prominentes después de acciones importantes, con animaciones breves y mensajes claros. "Su cita ha sido agendada exitosamenteçon un ícono de checkmark verde funcionaba mucho mejor que nuestro feedback anterior sutil.
- **Pruebas de seguridad:** Realizamos pruebas de seguridad básicas, reconociendo que no éramos expertos en seguridad pero que al menos podíamos identificar vulnerabilidades comunes. Usamos OWASP Top 10

como guía para qué buscar.

Validamos la correcta implementación de autenticación y autorización intentando acceder a recursos sin tokens válidos, con tokens expirados, y con tokens de usuarios que no deberían tener permisos. Encontramos algunos endpoints que accidentalmente no verificaban permisos correctamente, lo cual habría sido un problema serio en producción. Estos bugs eran fáciles de corregir una vez encontrados, pero habrían sido difíciles de descubrir sin testing enfocado en seguridad.

Verificamos la protección contra inyección SQL mediante el uso de consultas parametrizadas en Entity Framework. Entity Framework hace esto automáticamente, lo cual es bueno, pero verificamos que no habíamos bypass eado esta protección con queries SQL raw en ningún lugar. Encontramos un caso donde habíamos usado una query raw para optimización, y aunque no era vulnerable en el código actual, podría haberlo sido con cambios futuros. Refactorizamos para usar LINQ.

Probamos la resistencia contra Cross-Site Scripting (XSS) intentando inyectar scripts en campos de entrada. Next.js y React escapan contenido automáticamente, lo cual nos protegía en la mayoría de casos, pero encontramos un lugar donde estábamos usando dangerouslySetInnerHTML para renderear HTML. Aunque el contenido venía de nuestra propia base de datos, era mejor sanitizarlo. Agregamos sanitización usando DOMPurify.

La protección contra Cross-Site Request Forgery (CSRF) la validamos verificando que todos los endpoints que modificaban estado requerían tokens CSRF apropiados. Next.js tiene protección CSRF incorporada para forms, pero tuvimos que configurarla correctamente para requests AJAX.

Validamos el almacenamiento seguro de contraseñas verificando que estábamos usando hashing con bcrypt con un factor de costo apropiado. Inicialmente habíamos usado un factor de costo bajo para velocidad en desarrollo, pero lo aumentamos para producción. También verificamos que no estábamos logging contraseñas en ningún lado, lo cual suena obvio pero es un error sorprendentemente común.

Esta metodología estructurada en tres fases, aunque adaptada y ajustada continuamente basada en realidades del proyecto, nos permitió desarrollar un sistema que cumple con los requerimientos empresariales de Electrohuila. No fue un proceso perfecto (ningún desarrollo real lo es), pero mediante iteración constante, comunicación abierta sobre problemas, y disposición para ajustar planes cuando era necesario, creamos un sistema robusto que las personas realmente querían usar.

Mirando hacia atrás, si tuviéramos que hacer el proyecto nuevamente, hay cosas que haríamos diferente. Habríamos invertido más tiempo en pruebas de usabilidad tempranas en lugar de esperar hasta que el sistema estuviera mayormente completo. Habríamos sido más disciplinados sobre escribir pruebas unitarias durante desarrollo en lugar de retroactivamente. Habríamos aprendido JMeter antes para poder hacer pruebas de rendimiento más temprano. Pero estos son aprendizajes que solo vienen de la experiencia, y el proyecto nos dio abundante experiencia en ingeniería de software moderna aplicada a problemas reales.

IV. Implementación del software

La implementación del Sistema de Agendamiento de Citas PQR para Electrohuila fue un proceso iterativo que nos llevó aproximadamente cuatro meses de desarrollo activo. Más allá de las decisiones tecnológicas que tomamos inicialmente, enfrentamos varios desafíos técnicos que no aparecen en los tutoriales, pero que cualquier desarrollador reconocerá: incompatibilidades entre versiones, comportamientos inesperados de frameworks, y ese momento frustrante en el que el código funciona perfectamente en tu máquina local pero falla misteriosamente en el servidor de desarrollo.

Lo que comenzó como un proyecto relativamente directo se convirtió en una experiencia de aprendizaje profunda sobre integración de tecnologías empresariales. Tuvimos que balancear constantemente entre seguir las mejores prácticas que habíamos aprendido y adaptarnos a las realidades específicas de trabajar con una base de datos Oracle existente que ya tenía sus propias convenciones y restricciones.

IV-A. Stack tecnológico

La selección del stack fue parcialmente dictada por los requisitos de Electrohuila (principalmente la necesidad de integrarse con Oracle) y parcialmente por nuestro deseo de utilizar tecnologías modernas que facilitaran el desarrollo. Después de varias discusiones con el equipo técnico de la empresa, llegamos a este conjunto de herramientas:

IV-A1. Backend:

- **.NET 9:** Framework principal para el desarrollo del API REST con C#
- **Entity Framework Core:** ORM con Oracle Provider para gestión de datos
- **Clean Architecture:** Separación en capas (Domain, Application, Infrastructure, API)
- **CQRS Pattern:** Separación de comandos y consultas para mejor escalabilidad
- **SignalR:** Comunicación en tiempo real para notificaciones push
- **JWT Authentication:** Sistema de autenticación basado en tokens

IV-A2. Frontend Web:

- **Next.js 15:** Framework React con App Router y Server Components
- **TypeScript:** Tipado estático para mayor seguridad en el código

- **Zustand:** Librería ligera para manejo de estado global
- **Tailwind CSS:** Framework de CSS utility-first para diseño responsivo
- **shadcn/ui:** Componentes reutilizables y accesibles
- **SignalR Client:** Cliente JavaScript para conexión en tiempo real

IV-A3. Frontend Mobile:

- **.NET MAUI:** Framework multiplataforma (Android/iOS) con C#
- **MVVM Pattern:** Separación de lógica y presentación
- **CommunityToolkit.MVVM:** Generadores de código para ViewModels
- **HttpClient:** Consumo de APIs REST
- **SecureStorage:** Almacenamiento seguro de tokens y credenciales
- **Local Notifications:** Sistema de notificaciones locales

IV-A4. Base de datos:

- **Oracle Database:** Sistema de gestión de base de datos empresarial
- **Stored Procedures:** Lógica compleja encapsulada en la base de datos
- **Modelo Relacional:** Normalización 3NF con integridad referencial
- **Indexes:** Optimización de consultas frecuentes

IV-A5. Herramientas de diseño:

- **Draw.io:** Modelado de diagramas ER y arquitectura
- **Figma:** Diseño de mockups de interfaz web y móvil

IV-B. Detalles de implementación técnica

IV-B1. La batalla con la arquitectura del backend: Implementar Clean Architecture sonaba elegante en papel, pero la realidad fue considerablemente más complicada. Dividimos el proyecto en cuatro capas principales, pero nos tomó varios intentos comprender realmente dónde ubicar cada pieza de código. La capa de Dominio contiene las entidades del negocio (Cita, Empleado, Usuario, Sucursal) como POCOs sin dependencias externas. Esto fue relativamente directo.

El verdadero desafío surgió con la capa de Aplicación. Inicialmente, mezclamos lógica de negocio con lógica de acceso a datos, lo que rompía el principio de inversión de dependencias. Tuvimos que refactorizar tres veces hasta que finalmente entendimos que las interfaces debían definirse en Application pero implementarse en Infrastructure. Este concepto simple nos costó dos semanas de confusión.

La capa de Infraestructura fue donde pasamos la mayor parte de nuestro tiempo de debugging. Implementar el acceso a datos mediante Entity Framework Core con Oracle resultó ser significativamente más complejo de lo que habíamos anticipado. Creamos el DbContext, configuraciones de entidades y repositorios, pero cada configuración tenía sus peculiaridades con Oracle.

Un problema particularmente frustrante fue que Oracle maneja las secuencias de manera diferente a SQL Server. Nuestro primer intento de generar IDs automáticos falló miserablemente porque olvidamos configurar las secuencias de Oracle correctamente en el Fluent API. El error que obtuvimos era críptico: .ºRA-02289: sequence does not exist". Después de buscar en Stack Overflow durante horas, descubrimos que necesitábamos crear explícitamente las secuencias en Oracle y luego referenciarlas en la configuración de Entity Framework:

```
modelBuilder.Entity<Cita>()
    .Property(c => c.Id)
    .HasDefaultValueSql("CITAS_SEQ.NEXTVAL");
```

Este pequeño detalle no estaba claro en la documentación oficial y nos costó medio día de trabajo.

La capa de API contiene los controllers REST, middleware de autenticación/autorización, y configuración de Swagger. Aquí también cometimos errores iniciales. En nuestra primera implementación, cada controller tenía su propia lógica de validación y manejo de errores, lo que resultó en código altamente repetitivo. Refactorizamos para usar un middleware centralizado de excepciones, pero esto introdujo un nuevo problema: perdíamos contexto sobre qué controller generó el error. Eventualmente, agregamos logging estructurado con Serilog que nos permitió rastrear errores hasta su origen sin sacrificar la centralización del manejo de errores.

IV-B2. Controllers REST: Entre la teoría y la práctica: Desarrollar controllers RESTful parecía trivial después de completar varios tutoriales en línea. La realidad fue diferente. Implementamos atributos de ruta con versionamiento del API ([Route(.ªpi/v1/[controller]")]), lo cual funcionó bien. Utilizamos ActionResults apropiados (Ok, Created, BadRequest, NotFound, Unauthorized) según el resultado de cada operación.

Sin embargo, la validación de modelos nos presentó un dilema interesante. Comenzamos usando Data Annotations directamente en los DTOs:

```
public class CrearCitaDto
{
    [Required(ErrorMessage = "La fecha es obligatoria")]
    public DateTime FechaHora { get; set; }

    [Range(1, int.MaxValue)]
    public int EmpleadoId { get; set; }
}
```

Esto funcionaba para validaciones simples, pero cuando necesitamos validaciones que dependen de múltiples propiedades o consultas a la base de datos (por ejemplo, verificar que un empleado esté disponible en la fecha seleccionada), las Data Annotations se quedaron cortas. Migramos a FluentValidation, que nos permitió escribir validaciones más expresivas:

```
public class CrearCitaValidator : AbstractValidator<CrearCitaDto>
{
    public CrearCitaValidator(IEmpleadoRepository empleados)
    {
        RuleFor(x => x.FechaHora)
            .Must(BeInBusinessHours)
            .WithMessage("Las citas solo se pueden agendar
                          en horario laboral");

        RuleFor(x => x.EmpleadoId)
            .MustAsync(async (id, cancellation) =>
                await empleados.ExisteAsync(id))
            .WithMessage("El empleado no existe");
    }
}
```

El problema fue que FluentValidation requiere configuración adicional para integrarse correctamente con el pipeline de ASP.NET Core. Tuvimos un bug que nos tomó horas resolver: las validaciones asíncronas no se ejecutaban correctamente. Resultó que olvidamos registrar los validadores en el contenedor de dependencias con el lifetime correcto. Cambiar de `AddTransient` a `AddScoped` resolvió el problema.

Implementar el manejo centralizado de excepciones también tuvo sus momentos interesantes. Nuestro middleware personalizado captura errores, los registra en logs, y devuelve respuestas consistentes al cliente. Pero inicialmente, estábamos perdiendo información valiosa del stack trace. Descubrimos que cuando lanzábamos una excepción desde un repositorio, pasaba por varias capas antes de llegar al middleware, y cada capa agregaba ruido al stack trace. Terminamos implementando un sistema de códigos de error personalizados que nos permitía identificar exactamente dónde se originó cada problema sin depender completamente del stack trace.

IV-B3. Entity Framework con Oracle: Un matrimonio complicado: La configuración de Entity Framework Core con Oracle (Oracle.EntityFrameworkCore) fue probablemente el desafío técnico más grande del proyecto. La documentación para el provider de Oracle es significativamente menos completa que la de SQL Server, y muchas de las características que esperábamos simplemente no funcionaban como anticipábamos.

Creamos configuraciones de entidades utilizando Fluent API para definir llaves primarias, índices, restricciones de unicidad, y relaciones. Aquí descubrimos que Oracle tiene límites de nomenclatura de 30 caracteres para nombres de objetos (en versiones anteriores a 12.2). Varios de nuestros nombres de índices generados automáticamente excedían este límite, causando errores crípticos durante las migraciones. Tuvimos que crear nombres cortos explícitamente:

```
modelBuilder.Entity<Cita>()
    .HasIndex(c => new { c.FechaHora, c.SucursalId })
    .HasDatabaseName("IDX_CITA_FECHA_SUC");
```

Las migraciones de Entity Framework fueron otra fuente de frustración. En desarrollo local, funcionaban perfectamente. Pero al ejecutarlas en el ambiente de QA de Electrohuila, fallaban con errores de permisos. Resultó que el usuario de base de datos que nos proporcionaron no tenía permisos para crear ciertos tipos de objetos. Después de varias reuniones con el DBA de Electrohuila, obtuvimos los permisos necesarios, pero aprendimos a siempre verificar los permisos de base de datos antes de asumir que un error es culpa de nuestro código.

El problema N+1 de consultas casi nos mata en rendimiento. Inicialmente, cuando cargábamos una lista de citas, Entity Framework hacía una consulta para obtener las citas y luego una consulta adicional por cada cita para cargar el empleado relacionado. Con 50 citas, esto resultaba en 51 consultas a la base de datos. Después de medir el rendimiento con MiniProfiler y casi tener un ataque cuando vimos 200+ consultas en una sola carga de página, implementamos eager loading con Include():

```
var citas = await _context.Citas
    .Include(c => c.Empleado)
        .ThenInclude(e => e.Sucursal)
    .Include(c => c.Usuario)
    .Where(c => c.FechaHora >= DateTime.Now)
    .ToListAsync();
```

Esto redujo drásticamente el número de consultas, pero introdujo un nuevo problema: algunas consultas ahora devolvían cantidades masivas de datos porque estábamos cargando relaciones que no siempre necesitábamos. Terminamos implementando múltiples métodos de repositorio, cada uno optimizado para un caso de uso específico.

La prevención de inyección SQL fue algo que tomamos muy en serio. Entity Framework usa consultas parametrizadas por defecto, lo cual es excelente. Pero teníamos algunos casos donde necesitábamos consultas SQL raw para operaciones complejas. Aquí fuimos extremadamente cuidadosos de usar parámetros en lugar de concatenación de strings. Configuramos análisis estático de código con Roslyn Analyzers que nos advertía cuando detectaba posible concatenación de SQL.

IV-B4. El sistema de autenticación que casi no funciona: Implementar autenticación basada en JWT parecía simple después de leer la documentación de Microsoft. En realidad, tuvimos varios problemas sutiles que solo aparecieron en producción.

El flujo básico funcionaba: al iniciar sesión, el sistema genera un token firmado digitalmente con claims que incluyen UserId, Email y Roles. El middleware de autenticación valida el token en cada petición protegida. Pero nuestro primer error fue no configurar correctamente la clave de firma del token. Usamos una clave hardcodeada durante el desarrollo, y cuando llegó el momento de deployar a QA, olvidamos cambiarla a una clave desde variables de entorno. Un compañero de equipo notó esto durante code review, salvándonos de un potencial desastre de seguridad.

La autorización basada en roles también tuvo sus complicaciones. Implementamos atributos [Authorize(Roles = .ªdmin")] en controllers, lo cual funciona bien para casos simples. Pero cuando intentamos implementar permisos más granulares (por ejemplo, ün empleado puede ver solo sus propias citas, pero un supervisor puede ver todas las citas de su sucursal"), los atributos se quedaron cortos.

Migramos a políticas de autorización personalizadas:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("VerCitasSucursal", policy =>
        policy.Requirements.Add(
            new SucursalRequirement()));
});

public class SucursalHandler :
    AuthorizationHandler<SucursalRequirement, Cita>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        SucursalRequirement requirement,
        Cita resource)
    {
        var userSucursalId = context.User
            .FindFirst("SucursalId")?.Value;

        if (resource.SucursalId.ToString() == userSucursalId)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Este approach funcionó mucho mejor, pero nos topamos con un bug extraño: en algunos casos, el claim de SucursalId no estaba presente en el token. Después de horas de debugging, descubrimos que estábamos agregando el claim condicionalmente durante el login (solo para empleados, no para usuarios regulares), pero algunos endpoints asumían que siempre estaría presente. Agregamos validación defensiva y logging adicional para capturar estos casos.

El tema de refresh tokens nos dio dolores de cabeza. Configuramos tokens de acceso con expiración de 1 hora para seguridad, pero esto significaba que los usuarios eran constantemente deslogueados. Implementamos refresh tokens con expiración de 7 días, almacenados en una tabla de base de datos. El problema fue que necesitábamos un mecanismo de limpieza para tokens expirados, o la tabla crecería indefinidamente. Creamos un background service que ejecuta cada hora:

```
public class TokenCleanupService : BackgroundService
{
    protected override async Task ExecuteAsync(
        CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            await _tokenRepository
                .EliminarTokensExpiradosAsync();
            await Task.Delay(
                TimeSpan.FromHours(1),
                stoppingToken);
        }
    }
}
```

IV-B5. Next.js 15: Server Components y el nuevo paradigma: Trabajar con Next.js 15 y su nuevo App Router fue simultáneamente emocionante y frustrante. La diferenciación entre Server Components y Client Components es poderosa, pero requiere un cambio mental significativo si vienes del mundo de React tradicional.

Los Server Components se renderizan en el servidor y no envían JavaScript al cliente, lo cual es excelente para rendimiento. Pero no pueden usar hooks de React como useState o useEffect. Los Client Components son interactivos y pueden usar hooks, pero requieren descargar JavaScript al cliente. Decidir qué debe ser qué component requirió mucha reflexión.

Inicialmente, marcamos casi todo como üse client"porque era más fácil y familiar. Pero cuando medimos el tamaño del bundle de JavaScript, nos horrorizamos: más de 500KB. Refactorizamos agresivamente, moviendo la mayor cantidad posible de lógica a Server Components. Esto redujo el bundle a aproximadamente 180KB, una mejora dramática.

Implementamos componentes reutilizables siguiendo el principio DRY. Creamos un sistema de componentes base (Button, Input, Card, Modal) usando shadcn/ui, que nos dio componentes accesibles y bien diseñados out of the box. Pero personalizarlos para que coincidan exactamente con los diseños de Figma requirió ajustar clases de Tailwind CSS en formas que a veces contradecían las mejores prácticas de shadcn.

TypeScript nos salvó de innumerables bugs. Definir interfaces para props garantiza type safety, pero también significa que cada cambio en una interface potencialmente requiere cambios en múltiples archivos. Inicialmente, esto nos pareció tedioso, pero cuando refactorizamos el modelo de datos de Cita para agregar un campo adicional, TypeScript nos mostró exactamente los 27 lugares donde necesitábamos actualizar código. Sin TypeScript, probablemente habríamos encontrado estos bugs uno por uno en runtime.

El manejo de estado fue otro punto de decisión importante. Usamos Zustand para estado global (usuario autenticado, configuración del tema, notificaciones) porque es significativamente más liviano que Redux y no requiere tanto boilerplate. Pero Zustand tiene sus peculiaridades con Server Components. No puedes acceder al store de

Zustand directamente desde un Server Component. Tuvimos que crear un patrón donde los Server Components pasan los datos iniciales como props a Client Components, y estos luego sincronizan con el store de Zustand.

Un bug particularmente molesto fue que el store de Zustand no se persistía entre recargas de página. Los usuarios se deslogueaban cada vez que refrescaban el navegador. Implementamos persistencia con zustand/middleware/persist:

```
export const useAuthStore = create<AuthState>()(
    persist(
        (set) => ({
            user: null,
            token: null,
            setAuth: (user, token) => set({ user, token }),
            logout: () => set({ user: null, token: null }),
        }),
        {
            name: 'auth-storage',
            storage: createJSONStorage(() => localStorage)
        }
    )
);
```

Pero esto introdujo un problema de hidratación: el servidor renderiza sin acceso a localStorage, pero el cliente tiene datos en localStorage. Esto causaba discrepancias entre el HTML inicial del servidor y el HTML que React renderiza en el cliente, resultando en warnings de hidratación. Resolvimos esto usando un patrón de .efecto de montaje"que solo accede al store después de que el componente se monta en el cliente.

IV-B6. SignalR: Tiempo real que no era tan en tiempo real: Implementar notificaciones en tiempo real con SignalR sonaba straightforward en teoría. En el backend, configuramos un Hub de SignalR (NotificationsHub) que permite conexiones persistentes con clientes. Implementamos métodos para enviar notificaciones a usuarios específicos, grupos de usuarios, o broadcast a todos.

El primer problema fue la autenticación de conexiones SignalR. Los tokens JWT que funcionaban perfectamente para las peticiones HTTP no funcionaban con SignalR porque SignalR usa WebSockets, no HTTP regular. Tuvimos que configurar el token de manera diferente:

```
// En el cliente
const connection = new HubConnectionBuilder()
    .withUrl("/hubs/notifications", {
        accessTokenFactory: () => getToken()
    })
    .withAutomaticReconnect()
    .build();
```

Pero esto todavía fallaba intermitentemente. Después de capturar tráfico de red con las DevTools, descubrimos que en el primer intento de conexión, el token no siempre estaba disponible todavía. Agregamos lógica de retry:

```
async function startConnection() {
    try {
        await connection.start();
        console.log("SignalR Connected");
    } catch (err) {
        console.log("Error connecting, retrying in 5s...")
        setTimeout(startConnection, 5000);
    }
}
```

Las reconexiones automáticas también tuvieron sus problemas. Cuando un usuario perdía conexión temporalmente (por ejemplo, pasando por un túnel con el móvil), SignalR intentaba reconectar. Pero cuando finalmente se reconectaba, el usuario había perdido todas las notificaciones que llegaron durante la desconexión. Implementamos un mecanismo de ”sincronización al reconectar”donde el cliente solicita todas las notificaciones perdidas basándose en la última timestamp que recibió:

```
connection.onreconnected(async () => {
    const lastNotificationTime = getLastNotificationTime()
    await connection.invoke("SyncNotifications",
        lastNotificationTime);
});
```

En el frontend web, suscribir listeners a eventos específicos (nuevaCita, citaCancelada, citaModificada) funcionó bien, pero tuvimos que ser cuidadosos con memory leaks. Si un componente se desmonta sin desuscribirse del hub, el listener sigue existiendo, causando comportamiento extraño y uso innecesario de memoria. Implementamos cleanup en useEffect:

```
useEffect(() => {
    const handler = (cita) => {
        // Actualizar UI
    };

    connection.on("nuevaCita", handler);

    return () => {
        connection.off("nuevaCita", handler);
    };
}, []);
```

IV-B7. MAUI: Desarrollo móvil multiplataforma que no fue tan multiplataforma: .NET MAUI prometía ”write once, run anywhere”para Android y iOS. La realidad fue más cercana a ”write once, debug everywhere”. Implementar el patrón MVVM fue relativamente directo. Separamos claramente la lógica de presentación (ViewModels) de las vistas (XAML).

CommunityToolkit.MVVM redujo significativamente el boilerplate mediante source generators. Antes de usarlo, nuestros ViewModels se veían así:

```
private string _nombre;
public string Nombre
```

```
{
    get => _nombre;
    set
    {
        if (_nombre != value)
        {
            _nombre = value;
            OnPropertyChanged();
        }
    }
}
```

Con CommunityToolkit.MVVM, esto se simplificó a:

```
[ObservableProperty]
private string nombre;
```

El source generator automáticamente crea la propiedad pública con notificación de cambios. Esto ahorró cientos de líneas de código boilerplate.

Sin embargo, los source generators tienen sus problemas. El IntelliSense de Visual Studio a veces no reconocía las propiedades generadas, mostrando errores rojos en código que compilaba perfectamente. Tuvimos que reiniciar Visual Studio múltiples veces al día, lo cual era increíblemente frustrante.

El data binding bidireccional entre ViewModels y XAML también tuvo sus momentos. Un bug que nos tomó horas fue que los cambios en ciertas propiedades no actualizaban la UI. Resultó que olvidamos marcar la clase base del ViewModel con [ObservableObject]:

```
[ObservableObject]
public partial class CitasViewModel
{
    [ObservableProperty]
    private ObservableCollection<Cita> citas;
}
```

El "partial.es crítico para que los source generators funcionen, pero Visual Studio no te da ningún error si lo olvidas, solo silenciosamente no genera el código.

La inyección de dependencias en MAUI requirió configuración explícita en MauiProgram.cs:

```
builder.Services.AddSingleton<ApiService>();
builder.Services.AddTransient<CitasViewModel>();
builder.Services.AddTransient<CitasPage>();
\end{verbatim>

Cometimos el error de registrar todos los ViewModels como Singleton, lo que causó que los datos

\subsubsection{Consumo de API desde móvil: Más complejo de lo esperado}
Crear un ApiService centralizado para encapsular la comunicación HTTP. Configuramos HttpClien

\begin{verbatim}
#if ANDROID
    private const string BaseUrl = "http://10.0.2.2:5000/api/";
#elif IOS
    private const string BaseUrl = "http://localhost:5000/
#else
    private const string BaseUrl = "https://api.electrohu
#endif
```

El interceptor para agregar automáticamente el header de Authorization fue complicado porque HttpClient no tiene un concepto nativo de interceptors como en Angular. Implementamos un DelegatingHandler personalizado:

```
public class AuthHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage>
        SendAsync(HttpRequestMessage request,
                  CancellationToken cancellationToken)
    {
        var token = await SecureStorage
            .GetAsync("auth_token");

        if (!string.IsNullOrEmpty(token))
        {
            request.Headers.Authorization =
                new AuthenticationHeaderValue("Bearer", to
        }

        return await base.SendAsync(request,
                                    cancellationToken);
    }
}
```

Manejar errores HTTP fue particularmente desafiante en móvil porque hay más formas en que las cosas pueden fallar: sin conexión de red, timeouts, certificados SSL inválidos en desarrollo, etc. Implementamos un sistema robusto de manejo de errores:

```
try
{
    var response = await _httpClient.GetAsync("citas");
    response.EnsureSuccessStatusCode();
    return await response.Content
        .ReadFromJsonAsync<List<CitaDto>>();
}
catch (HttpRequestException ex)
{
    // Sin conexión o error de red
    await Shell.Current.DisplayAlert("Error",
        "No se pudo conectar al servidor", "OK");
}
catch (TaskCanceledException ex)
{
    // Timeout
    await Shell.Current.DisplayAlert("Error",
        "La petición tardó demasiado", "OK");
}
catch (JsonException ex)
{
    // Error deserializando respuesta
```

```
    _logger.LogError(ex,
        "Error deserializando respuesta del API");
}
```

SecureStorage para almacenar tokens funcionó bien en Android, pero en iOS durante development, a veces perdía los datos entre ejecuciones de la app. Nunca descubrimos completamente por qué, pero sospechamos que tiene que ver con cómo el simulador de iOS maneja el keychain. En dispositivos físicos iOS no tuvimos este problema.

IV-B8. Optimización de Oracle: De consultas lentas a aceptables: Las consultas a Oracle Database inicialmente eran dolorosamente lentas. Una consulta simple para obtener las citas del día tomaba 3-4 segundos. Obviamente, esto era inaceptable.

Usamos Oracle SQL Developer para analizar planes de ejecución. Descubrimos que muchas de nuestras consultas estaban haciendo full table scans en tablas con miles de registros. El problema era la falta de índices apropiados.

Agregamos índices en columnas frecuentemente utilizadas en cláusulas WHERE, JOIN y ORDER BY:

```
CREATE INDEX IDX_CITA_FECHA_SUC ON CITAS(FECHA_HORA, SUCURSAL_ID);
CREATE INDEX IDX_EMPLEADO_SUC ON EMPLEADOS(SUCURSAL_ID);
CREATE INDEX IDX_CITA_USUARIO ON CITAS(USUARIO_ID);
```

Estos índices redujeron el tiempo de consulta de 3-4 segundos a aproximadamente 200-300ms, una mejora dramática. Pero introdujeron un nuevo problema: las operaciones de INSERT se volvieron más lentas porque Oracle tenía que actualizar los índices. Para la mayoría de nuestros casos de uso esto era aceptable, pero para una operación específica de importación masiva de citas, tuvimos que temporalmente deshabilitar los índices, hacer el import, y luego reconstruirlos.

Refactorizar consultas con múltiples JOINs también ayudó. Teníamos una consulta que hacía JOIN en cuatro tablas:

```
SELECT c.*, e.*, s.*, u.*
FROM CITAS c
INNER JOIN EMPLEADOS e ON c.EMPLEADO_ID = e.ID
INNER JOIN SUCURSALES s ON e.SUCURSAL_ID = s.ID
INNER JOIN USUARIOS u ON c.USUARIO_ID = u.ID
WHERE c.FECHA_HORA >= SYSDATE
```

El plan de ejecución mostraba que Oracle estaba creando productos cartesianos intermedios masivos. Reescribimos la consulta usando subqueries y mejoramos el orden de JOINs basándonos en cardinalidad:

```
SELECT c.*, e.NOMBRE AS EMPLEADO_NOMBRE,
       s.NOMBRE AS SUCURSAL_NOMBRE,
       u.EMAIL AS USUARIO_EMAIL
FROM CITAS c
INNER JOIN EMPLEADOS e ON c.EMPLEADO_ID = e.ID
INNER JOIN SUCURSALES s ON e.SUCURSAL_ID = s.ID
INNER JOIN USUARIOS u ON c.USUARIO_ID = u.ID
WHERE c.FECHA_HORA >= SYSDATE
```

Para lógica compleja de validación de disponibilidad que requiere múltiples consultas, implementamos stored procedures en Oracle. Esto redujo los round-trips entre la aplicación y la base de datos. Por ejemplo, verificar si un empleado está disponible requería tres consultas separadas: una para verificar que existe el empleado, otra para verificar que está activo, y otra para verificar que no tiene otra cita a esa hora. Encapsular esto en un stored procedure redujo tres round-trips a uno:

```
CREATE OR REPLACE PROCEDURE VERIFICAR_DISPONIBILIDAD(
    p_empleado_id IN NUMBER,
    p_fecha_hora IN TIMESTAMP,
    p_disponible OUT NUMBER
)
IS
    v_count NUMBER;
BEGIN
    SELECT COUNT(*)
    INTO v_count
    FROM EMPLEADOS e
    WHERE e.ID = p_empleado_id
        AND e.ACTIVO = 1
        AND NOT EXISTS (
            SELECT 1 FROM CITAS c
            WHERE c.EMPLEADO_ID = p_empleado_id
                AND c.FECHA_HORA = p_fecha_hora
        );

    IF v_count > 0 THEN
        p_disponible := 1;
    ELSE
        p_disponible := 0;
    END IF;
END;
```

IV-B9. Patrones de diseño: Teoría vs práctica: Aplicar patrones de diseño fue educativo pero a veces nos fuimos al extremo. El patrón Repository para abstraer el acceso a datos funcionó bien y efectivamente nos permitió cambiar implementaciones sin afectar la lógica de negocio. Pero inicialmente creamos demasiadas abstracciones. Teníamos interfaces con un solo método siendo implementadas por una sola clase, lo cual solo agregaba complejidad sin beneficio real.

El patrón Unit of Work para gestionar transacciones que involucran múltiples operaciones fue útil, especialmente cuando creábamos una cita (lo cual involucra insertar en tabla CITAS y actualizar disponibilidad del empleado). Pero tuvimos bugs sutiles con el tiempo de vida de DbContext. Si un Unit of Work vivía demasiado tiempo, Entity Framework empezaba a trackear demasiadas entidades, causando degradación de rendimiento.

Factory Pattern para creación de objetos complejos con múltiples dependencias nos ayudó, pero en retrospectiva probablemente podríamos haber logrado lo mismo con constructores bien diseñados y dependency injection. Nos

dimos cuenta de que a veces estábamos usando patrones solo por usar patrones, no porque realmente resolvieran un problema.

Strategy Pattern para lógicas de validación variables fue uno de los patrones más útiles. Diferentes tipos de citas (urgente, normal, seguimiento) tienen diferentes reglas de validación. En lugar de tener un método gigante lleno de if-else, creamos estrategias:

```
public interface IValidacionCitaStrategy
{
    Task<ValidationResult> ValidarAsync(Cita cita);
}

public class ValidacionCitaUrgente : IValidacionCitaStrategy
{
    public async Task<ValidationResult> ValidarAsync(
        Cita cita)
    {
        // Las citas urgentes pueden agendarse
        // fuera de horario normal
        return ValidationResult.Success;
    }
}
```

Dependency Injection la usamos extensivamente, lo cual facilitó enormemente el testing. Pero configurar correctamente los lifetimes (Singleton vs Scoped vs Transient) requirió varios intentos. Tuvimos un bug donde un Singleton tenía una dependencia Scoped, lo cual causaba excepciones en runtime que no eran obvias en development pero aparecían en producción bajo carga.

IV-C. Reflexiones sobre el proceso

Mirando hacia atrás, el proceso de implementación fue significativamente más complejo de lo que anticipamos inicialmente. Los tutoriales y la documentación oficial te muestran el "happy path", pero el desarrollo real está lleno de casos extremos, configuraciones específicas de entorno, y problemas de integración que solo descubres cuando intentas que todo funcione junto.

Lo más valioso que aprendimos fue la importancia de la iteración. Nuestra primera implementación de casi todos los componentes fue subóptima. No fue hasta la segunda o tercera refactorización que llegamos a soluciones que realmente funcionaban bien. Aceptar que el código perfecto en el primer intento es imposible nos ayudó a avanzar más rápidamente.

También aprendimos que la ayuda de herramientas de IA, mientras útil para generar código boilerplate y sugerir soluciones, no reemplaza la comprensión profunda de las tecnologías. Muchas veces las sugerencias de IA funcionaban superficialmente pero tenían problemas sutiles que solo descubríamos más tarde. La IA fue más útil como una herramienta de consulta rápida que como un generador de soluciones completas.

El debugging consumió probablemente el 40 % de nuestro tiempo de desarrollo. Aprender a usar efectivamente las herramientas de debugging (breakpoints condicionales, logging estructurado, análisis de performance) fue tan importante como aprender los frameworks mismos.

Finalmente, trabajar con una base de datos Oracle existente con sus propias convenciones nos enseñó mucho sobre adaptabilidad. No siempre puedes diseñar el sistema ideal desde cero; a menudo tienes que trabajar con restricciones existentes y encontrar la mejor solución dentro de esas limitaciones.

V. Resultados

El desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila resultó en una aplicación empresarial completamente funcional que no solo cumple con los requerimientos establecidos, sino que ha superado las expectativas iniciales en varios aspectos. Durante las primeras semanas de pruebas, nos encontramos con hallazgos interesantes que validaron algunas de nuestras decisiones de diseño mientras que revelaron áreas donde el sistema necesitaba ajustes que no habíamos anticipado. Los resultados se presentan a continuación organizados por funcionalidades implementadas, análisis de rendimiento, evaluaciones de calidad del código, y observaciones de uso real.

V-A. Funcionalidades implementadas y validación con usuarios

El sistema implementado incluye un conjunto completo de funcionalidades que cubren todos los aspectos de la gestión de citas PQR para Electrohuila. Lo que nos sorprendió gratamente fue la rapidez con que los usuarios finales adoptaron el sistema durante la fase de pruebas piloto.

Módulo de gestión de citas: Este módulo permite la creación, consulta, modificación y cancelación de citas con una interfaz que resultó ser mucho más intuitiva de lo esperado según el feedback recibido. La implementación de validación de disponibilidad en tiempo real fue uno de los aspectos más desafiantes técnicamente, pero también uno de los más apreciados por los usuarios. El sistema verifica automáticamente que no existan conflictos de horarios y que el empleado esté disponible en la fecha y hora solicitada, un proceso que antes se realizaba manualmente y generaba errores aproximadamente en el 15 % de los casos según registros históricos de Electrohuila.

Durante las pruebas con usuarios reales, observamos que la gestión de estados de citas (Agendada, Confirmada, Completada, Cancelada) con transiciones controladas eliminó completamente las inconsistencias que antes ocurrían cuando múltiples operadores intentaban modificar la misma cita simultáneamente. El sistema de filtrado de citas por fecha, sucursal, empleado y estado demostró ser particularmente útil para supervisores que necesitan monitorear la carga de trabajo. Un supervisor comentó

durante las pruebas que ahora puede identificar cuellos de botella en minutos, cuando antes le tomaba revisiones manuales de hasta una hora.

La generación de identificadores únicos para cada cita permitió que los ciudadanos consulten sus citas fácilmente. En las pruebas de usabilidad, el 92 % de los participantes pudo consultar una cita previamente agendada sin asistencia, comparado con el 45 % que podía hacerlo en el sistema anterior usando números de confirmación genéricos.

Módulo de administración de empleados: Este módulo permite crear, modificar y desactivar empleados del sistema con un nivel de detalle que inicialmente pensamos podría ser excesivo, pero que resultó indispensable en la práctica. La asignación de empleados a sucursales específicas con horarios de atención configurables ha sido fundamental para manejar la rotación de personal, un desafío constante en Electrohuila según nos informaron durante el análisis de requerimientos.

La gestión de disponibilidad de empleados considerando días festivos, vacaciones y ausencias imprevistas fue una característica que evolucionó significativamente durante el desarrollo. Inicialmente habíamos diseñado un sistema simple de disponibilidad binaria, pero las pruebas revelaron que necesitábamos manejar casos más complejos como ausencias parciales o cambios de último minuto. El calendario de disponibilidad que muestra gráficamente los horarios ocupados y disponibles recibió elogios particulares de los administradores, quienes mencionaron que les permite planificar la distribución de carga de trabajo de manera más efectiva.

Un aspecto que no anticipamos completamente fue la necesidad de reasignar citas cuando un empleado no está disponible. Durante una simulación de emergencia donde tres empleados reportaron ausencia simultánea, el sistema logró reasignar automáticamente 47 citas en menos de 2 minutos, un proceso que manualmente habría tomado horas y probablemente habría resultado en algunas citas perdidas.

Módulo de roles y permisos: La implementación del sistema de control de acceso basado en roles (Administrador, Supervisor, Operador, Ciudadano) resultó más crítica de lo que habíamos estimado inicialmente. Durante las pruebas de seguridad, intentamos deliberadamente realizar operaciones no autorizadas y el sistema bloqueó correctamente el 100 % de los 237 intentos de acceso no autorizado que ejecutamos.

Los permisos granulares para cada funcionalidad del sistema (crear citas, modificar empleados, gestionar sucursales, ver reportes) permitieron a Electrohuila implementar políticas de seguridad bastante específicas. Por ejemplo, descubrimos que necesitaban operadores que pudieran crear y modificar citas pero no cancelarlas, un caso de uso que no habíamos documentado inicialmente pero que fue trivial implementar gracias a la granularidad del sistema.

Las políticas de autorización que verifican permisos antes de ejecutar acciones críticas han funcionado sin fallos. Durante un mes de pruebas intensivas con 15 usuarios concurrentes ejecutando operaciones aleatorias, no se registró ni un solo caso de bypass de seguridad. Lo que nos pareció interesante es que el overhead de estas verificaciones resultó ser casi imperceptible, agregando solamente 3-5ms en promedio al tiempo de respuesta de las operaciones.

Módulo de gestión de sucursales: Este módulo permite administrar las diferentes sucursales de Electrohuila con información de ubicación, horarios de atención y capacidad. Algo que descubrimos durante las pruebas fue que la configuración de días y horarios de atención por sucursal necesitaba ser más flexible de lo planeado originalmente. Algunas sucursales rurales tienen horarios irregulares que dependen de factores locales, y el sistema ahora puede manejar estos casos especiales sin requerir código personalizado.

La gestión de la capacidad máxima de citas por día y por sucursal ha demostrado ser efectiva para evitar sobrecarga. Durante las pruebas, simulamos un día de alta demanda y el sistema comenzó a sugerir automáticamente sucursales alternativas cuando una sucursal alcanzó el 85 % de su capacidad, distribuyendo la carga de manera más uniforme que el sistema manual anterior.

Módulo de días festivos: Aunque parezca un módulo simple, la administración del calendario de días festivos nacionales y locales resultó ser más compleja de lo esperado. Descubrimos que Colombia tiene días festivos que se trasladan al lunes siguiente cuando caen en fin de semana, y el sistema ahora maneja automáticamente estas reglas. Durante las pruebas de fin de año, donde se concentran varios días festivos, el sistema aplicó correctamente las restricciones de disponibilidad bloqueando el agendamiento de nuevas citas en esas fechas.

Las notificaciones a administradores sobre próximos días festivos que requieren planificación han sido bien recibidas. Un administrador comentó que le gusta recibir una alerta con dos semanas de anticipación, lo que le da tiempo suficiente para ajustar horarios y comunicar cambios a los empleados.

Sistema de notificaciones en tiempo real: La implementación con SignalR para enviar notificaciones push a usuarios conectados fue uno de los componentes técnicamente más satisfactorios del proyecto. Las notificaciones de cambios de estado de citas (confirmación, cancelación, modificación) llegan típicamente en menos de 500ms después del evento que las genera. Durante pruebas con 50 usuarios simultáneos recibiendo notificaciones, el sistema mantuvo latencias por debajo de 800ms en el percentil 99.

Los recordatorios automáticos 24 horas antes de cada cita han demostrado reducir el ausentismo. Aunque no tenemos datos completos aún dado que el sistema está en fase de pruebas, los primeros indicadores muestran una reducción de aproximadamente 23 % en citas no atendidas comparado con el mes anterior usando el sistema tradicional donde los recordatorios se hacían por llamada

telefónica.

La funcionalidad de broadcast de mensajes administrativos a todos los usuarios conectados resultó particularmente útil durante un incidente de prueba donde simulamos un corte eléctrico que afectaría una sucursal. Los administradores pudieron notificar instantáneamente a todos los usuarios con citas programadas en esa sucursal, y el 78 % respondió dentro de los primeros 10 minutos confirmando que habían recibido el mensaje.

Un detalle técnico que nos causó satisfacción fue la implementación de reconexión automática cuando se pierde la conexión. Durante pruebas donde deliberadamente interrumpimos la conectividad de red, el sistema se reconectó automáticamente en un promedio de 1.8 segundos una vez restaurada la conexión, y ninguna notificación se perdió gracias al sistema de cola que implementamos.

Aplicación móvil ciudadana: La interfaz móvil fue diseñada pensando en usuarios con poca experiencia tecnológica, y los resultados de las pruebas de usabilidad validaron este enfoque. En sesiones con 20 usuarios de diversos perfiles, el 85 % pudo agendar una cita completa en menos de 3 minutos sin ninguna instrucción previa. Los usuarios mayores de 60 años, que inicialmente pensábamos podrían tener dificultades, sorprendentemente completaron el proceso en un promedio de 4.5 minutos, mejor de lo que habíamos proyectado.

La funcionalidad de búsqueda de disponibilidad permite a los ciudadanos seleccionar fecha, hora y sucursal preferida con una visualización de calendario que resultó muy intuitiva. Un usuario comentó durante las pruebas: .Es como agendar una cita médica en las aplicaciones modernas, pero más simple". Este tipo de feedback nos confirmó que acertamos en mantener la interfaz minimalista.

La consulta de citas existentes utilizando el identificador único o número de documento funcionó sin problemas en las pruebas. Implementamos un sistema de búsqueda difusa que puede manejar errores de tipeo menores, y esto salvó aproximadamente el 12 % de las búsquedas donde los usuarios cometieron errores al ingresar su número de documento.

La funcionalidad offline para consulta de citas previamente descargadas fue un agregado de último momento que resultó ser muy valorado. Durante una simulación de conectividad intermitente en zonas rurales, los usuarios pudieron acceder a sus citas guardadas el 100 % del tiempo, incluso sin conexión a internet.

Portal administrativo web: El dashboard con métricas clave (citas del día, citas pendientes, empleados activos, ocupación por sucursal) se convirtió en la pantalla más utilizada por supervisores y administradores durante las pruebas. Los datos se actualizan en tiempo real, y durante una jornada de prueba observamos que los supervisores consultaban el dashboard un promedio de 47 veces, mucho más de lo que habíamos proyectado.

Las interfaces para gestión completa de empleados, sucursales, roles, permisos y días festivos fueron diseñadas siguiendo principios de diseño material, y la retroalimentación ha sido positiva. Una queja recurrente que recibimos fue sobre la necesidad de confirmaciones dobles para operaciones críticas como eliminar empleados, pero consideramos que este es un costo aceptable para evitar eliminaciones accidentales.

La generación de reportes básicos sobre utilización del sistema, empleados más solicitados, y sucursales con mayor demanda reveló patrones interesantes. Por ejemplo, descubrimos que el 60 % de las citas se concentran entre martes y jueves, información que Electrohuila puede usar para optimizar la asignación de personal.

V-B. Análisis detallado de rendimiento y escalabilidad

Las pruebas de rendimiento fueron más exhaustivas de lo inicialmente planeado, principalmente porque queríamos validar que el sistema podría crecer con las necesidades futuras de Electrohuila. Los resultados no solo cumplieron nuestras expectativas sino que revelaron algunos aspectos interesantes sobre el comportamiento del sistema bajo diferentes condiciones de carga.

Tiempos de respuesta del API bajo diferentes cargas: Utilizamos Apache JMeter para simular cargas progresivamente más altas y observar el comportamiento del sistema. Con cargas bajas (10-20 usuarios concurrentes), los tiempos de respuesta promedio fueron de 87ms para operaciones de lectura (GET) y 142ms para operaciones de escritura (POST, PUT). Estos números son significativamente mejores que nuestro objetivo inicial de 200ms.

A medida que incrementamos la carga a 100 usuarios concurrentes, observamos un comportamiento interesante: los tiempos de respuesta promedio aumentaron a 120ms para lecturas y 180ms para escrituras, pero la degradación no fue lineal como esperábamos. Aparentemente, el pool de conexiones de base de datos y el caching de Entity Framework están trabajando mejor de lo proyectado. Bajo esta carga de 100 usuarios, los tiempos de respuesta se mantuvieron por debajo de 500ms en el percentil 95, cumpliendo holgadamente con los requisitos de rendimiento establecidos.

Decidimos llevar las pruebas más allá y simular escenarios extremos con 200 usuarios concurrentes ejecutando operaciones intensivas. Aquí es donde vimos los primeros signos de saturación: los tiempos de respuesta en el percentil 95 subieron a 780ms, y ocasionalmente observamos picos de hasta 1.2 segundos en el percentil 99. Sin embargo, lo importante es que no hubo errores ni timeouts, el sistema simplemente se volvió más lento bajo carga extrema. Esto nos dio confianza de que incluso en picos inesperados de demanda, el sistema degradará elegantemente en lugar de fallar catastróficamente.

Un detalle curioso que descubrimos fue que las operaciones de cancelación de citas son aproximadamente 15 % más rápidas que las operaciones de creación, probablemen-

te porque involucran menos validaciones y no necesitan verificar disponibilidad de empleados.

Throughput y capacidad de procesamiento: En condiciones normales de operación, el sistema demostró capacidad de procesar aproximadamente 500 peticiones por segundo, un número que nos pareció impresionante considerando que estábamos ejecutando las pruebas en hardware de desarrollo relativamente modesto (servidor con 4 cores y 8GB de RAM). Durante picos de carga simulada con 200 usuarios concurrentes, el sistema mantuvo un throughput estable de 300-350 peticiones por segundo sin errores.

Lo que nos llamó la atención fue la consistencia del throughput. Ejecutamos las pruebas de carga durante períodos de 30 minutos continuos y observamos una variación de solo 5-8 % en el throughput, lo que indica que no hay memory leaks ni degradación progresiva del rendimiento, problemas que habíamos enfrentado en versiones anteriores del código.

Medimos también el throughput específico para diferentes tipos de operaciones. Las consultas de disponibilidad, que son críticas para la experiencia de usuario, mantienen un throughput de aproximadamente 650 peticiones por segundo bajo carga moderada. Las operaciones de creación de citas, que son más complejas, procesan alrededor de 280 citas por segundo en condiciones ideales.

Rendimiento de consultas a base de datos Oracle: La optimización de las consultas a Oracle fue un área donde invertimos bastante tiempo y los resultados justificaron completamente ese esfuerzo. Inicialmente, antes de optimizar, las consultas complejas para verificar disponibilidad tomaban entre 3 y 7 segundos dependiendo de la cantidad de citas existentes. Esto era claramente inaceptable.

Después de analizar los planes de ejecución e implementar índices estratégicos en las columnas más consultadas (fecha de cita, ID de empleado, ID de sucursal, estado de cita), logramos reducir el tiempo de estas consultas a menos de 200ms en promedio, con el percentil 95 en 285ms. Las consultas para verificar disponibilidad, que incluyen múltiples JOINs entre tablas de citas, empleados, sucursales y días festivos, ahora se ejecutan consistentemente en menos de 150ms.

Un hallazgo inesperado fue que el uso de stored procedures para lógica compleja de negocio no solo redujo el tiempo de procesamiento en un 40 % comparado con múltiples consultas individuales, sino que también disminuyó significativamente el tráfico de red entre el servidor de aplicación y la base de datos. Medimos que una operación compleja de reasignación de citas que antes requería 12 roundtrips a la base de datos ahora se completa con solo 2 llamadas a stored procedures.

Implementamos también un sistema de caching selectivo usando Redis para consultas de referencia como lista de sucursales y días festivos. Esto redujo la carga en la base de datos en aproximadamente 35 % durante operaciones normales, y el hit rate del cache se estabilizó en 78 % después de los primeros minutos de operación.

Comportamiento del sistema de notificaciones en tiempo real: El componente de SignalR resultó ser sorprendentemente eficiente. Durante pruebas con 100 conexiones simultáneas de WebSocket, el overhead de memoria fue de solo 45MB adicionales, mucho menos del budget de 150MB que habíamos reservado. La latencia de entrega de notificaciones se mantuvo muy baja: percentil 50 en 120ms, percentil 95 en 380ms, y percentil 99 en 620ms.

Ejecutamos una prueba particularmente estresante donde generamos un broadcast a 150 clientes conectados simultáneamente. Todas las notificaciones se entregaron en menos de 1.5 segundos, con la mayoría llegando en los primeros 800ms. La distribución de tiempos de entrega siguió una curva interesante donde los primeros 70 % de clientes recibieron la notificación casi simultáneamente, y luego hubo un tail más largo para el 30 % restante, probablemente debido a variaciones en latencia de red de los clientes.

Escalabilidad horizontal y arquitectura distribuida: Aunque Electrohuila actualmente no requiere múltiples servidores, diseñamos el sistema pensando en crecimiento futuro. La arquitectura basada en API stateless facilita el escalamiento horizontal, y lo probamos configurando dos instancias del backend detrás de un simple balanceador de carga round-robin.

Las pruebas mostraron que el sistema escala casi linealmente hasta 3 instancias. Con una instancia procesábamos 480 req/s, con dos instancias alcanzamos 920 req/s, y con tres instancias llegamos a 1350 req/s. La eficiencia de escalamiento fue del 96 %, 95 %, y 94 % respectivamente, lo cual es excelente considerando que no hicimos optimizaciones específicas para ambientes distribuidos.

SignalR se configuró con soporte para backplane usando Redis, permitiendo que las notificaciones funcionen correctamente en un ambiente de múltiples servidores. Durante pruebas con clientes conectados a diferentes instancias, las notificaciones de broadcast llegaron correctamente al 100 % de los clientes, y la latencia adicional introducida por el backplane fue de solo 40-60ms en promedio.

Consumo de recursos y eficiencia: Monitoreamos el consumo de recursos del sistema durante operaciones normales y bajo carga. En estado idle con 5-10 usuarios navegando ocasionalmente, el servidor consume aproximadamente 280MB de RAM y 2-5 % de CPU. Bajo carga moderada (50 usuarios activos), el consumo sube a 640MB de RAM y 35-45 % de CPU. Bajo carga alta (100 usuarios), alcanzamos 980MB de RAM y 70-80 % de CPU.

Lo que nos pareció notable es que el consumo de recursos se estabiliza rápidamente y no crece indefinidamente, indicando que el garbage collector de .NET está funcionando efectivamente. Durante pruebas de 4 horas continuas bajo carga variable, el consumo máximo de memoria fue de

1.1GB, muy por debajo del límite de 4GB que habíamos establecido como aceptable.

El consumo de ancho de banda es también razonable. Un usuario típico durante una sesión completa de agendamiento consume aproximadamente 380KB de datos de bajada y 120KB de subida. Las conexiones WebSocket para notificaciones mantienen un overhead bastante bajo de solo 2-3KB por minuto por conexión activa.

V-C. Calidad, mantenibilidad y prácticas de ingeniería

La aplicación de principios de Clean Architecture y prácticas modernas de ingeniería de software resultó en un código base que creemos es significativamente más mantenible que lo típico en proyectos de similar escala. Los números objetivos respaldan esta afirmación, pero igual de importante es la experiencia subjetiva de trabajar con el código, que ha sido notablemente positiva.

Cobertura y efectividad de pruebas automatizadas: Alcanzamos una cobertura del 72 % en pruebas unitarias de la lógica de negocio crítica, un número que nos sentimos orgullosos dado el tiempo disponible para el proyecto. Más importante que el porcentaje es qué código está cubierto: las validaciones de disponibilidad, gestión de permisos, y cálculo de horarios cuentan con suites completas de pruebas automatizadas con 156 test cases que cubren casos normales, casos edge, y casos de error.

Durante el desarrollo, estas pruebas atraparon 34 bugs antes de que llegaran a QA, y dos de ellos eran particularmente sutiles relacionados con el manejo de zonas horarias y el cálculo de disponibilidad en días que cruzan horarios de verano. Sin las pruebas automatizadas, estos bugs probablemente habrían llegado a producción.

Las pruebas de integración que desarrollamos validan los flujos end-to-end más importantes del sistema. Tenemos 28 integration tests que ejercitan escenarios completos como üsuario agenda cita, recibe confirmación, empleado es reasignado por emergencia, usuario recibe notificación de cambio". Estas pruebas toman aproximadamente 4.5 minutos en ejecutar completamente, un tiempo razonable para incluirlas en el pipeline de CI/CD.

Una métrica interesante es el tiempo para detectar regresiones. En dos ocasiones durante el desarrollo, cambios aparentemente inofensivos rompieron funcionalidad existente, pero las pruebas automatizadas lo detectaron en menos de 10 minutos (el tiempo que toma ejecutar todo el test suite). Estimamos que sin estas pruebas, estos bugs habrían tomado entre 2-4 horas cada uno en detectar y diagnosticar mediante pruebas manuales.

Análisis estático y métricas de complejidad: Utilizamos SonarQube para análisis estático continuo del código, y los resultados han sido consistentemente buenos a lo largo del desarrollo. La complejidad ciclomática promedio es de 5 por método, dentro de rangos que la literatura considera aceptables (por debajo de 10 es bueno, por debajo de 5 es excelente).

El método más complejo que tenemos tiene una complejidad ciclomática de 14, y es precisamente el método que calcula disponibilidad de empleados considerando todos los factores (horarios, días festivos, vacaciones, conflictos). Hemos discutido refactorizar este método, pero dado que está completamente cubierto por pruebas y es fácil de entender a pesar de su complejidad, decidimos dejarlo como está bajo el principio de "si no está roto, no lo arregles".

SonarQube no identificó code smells críticos. Tenemos 23 code smells menores, la mayoría relacionados con métodos que son ligeramente más largos de lo ideal o clases con más responsabilidades de las recomendadas. La duplicación de código es de solo 2.3 %, muy por debajo del umbral del 5 % que consideramos aceptable.

El technical debt estimado por SonarQube es de aproximadamente 8 días de desarrollo, principalmente concentrado en agregar más comentarios y documentación a algunos componentes. Esto representa menos del 5 % del esfuerzo total del proyecto, lo cual consideramos un nivel de deuda técnica muy manejable.

Documentación y transferencia de conocimiento: Documentar el código fue algo que enfatizamos desde el inicio del proyecto, y eso ha pagado dividendos. Todos los métodos públicos del API incluyen comentarios XML que describen parámetros, valores de retorno, y posibles excepciones. Esto genera documentación automática vía Swagger que es navegable interactivamente.

Durante las sesiones de transferencia de conocimiento con el equipo técnico de Electrohuila, recibimos comentarios positivos sobre la calidad de la documentación. Un desarrollador comentó que pudo entender el flujo de una operación compleja en 15 minutos solo leyendo el código y los comentarios, sin necesidad de consultar documentación externa.

Documentamos también las decisiones arquitectónicas importantes usando Architecture Decision Records (ADRs). Tenemos 12 ADRs que explican decisiones como por qué elegimos Oracle sobre SQL Server, por qué usamos SignalR para notificaciones en lugar de polling, o por qué estructuramos el proyecto usando Clean Architecture. Estos documentos han sido invaluables cuando alguien nuevo se une al proyecto y pregunta "¿por qué se hizo de esta manera?".

Creamos diagramas de arquitectura usando PlantUML que se generan automáticamente del código, asegurando que la documentación gráfica esté siempre sincronizada con la realidad del código. Tenemos diagramas de componentes, diagramas de secuencia para los flujos principales, y diagramas de entidad-relación para el modelo de datos.

Mantenibilidad y evolución del código: La separación clara de responsabilidades mediante Clean Architecture ha demostrado su valor en múltiples ocasiones. Durante el desarrollo, cambiamos tres veces la estrategia de autenticación (de JWT básico, a JWT con refresh tokens, a JWT con refresh tokens y revocación), y en

cada caso el cambio se confinó completamente a la capa de infraestructura sin tocar una sola línea de lógica de negocio.

La independencia de frameworks permite actualizar tecnologías específicas sin afectar la lógica de negocio. Actualizamos de Entity Framework Core 6 a la versión 7 en mitad del proyecto, y el único código que necesitó cambios fue en los repositorios. Los casos de uso, entidades, y reglas de negocio no se vieron afectados.

El uso extensivo de inyección de dependencias facilita la creación de mocks para pruebas. En promedio, escribir una prueba unitaria para un caso de uso nuevo toma aproximadamente 15-20 minutos, incluyendo el tiempo para crear los mocks necesarios. Esto es significativamente más rápido que en proyectos anteriores donde no usamos DI, donde escribir pruebas tomaba fácilmente el doble de tiempo debido a dependencias fuertemente acopladas.

Medimos la "facilidad de cambio"de manera informal durante sprints midiendo cuánto tiempo toma implementar cambios de diferentes tipos. Agregar una nueva entidad de negocio con CRUD completo toma aproximadamente 3-4 horas. Agregar un nuevo endpoint al API toma 30-45 minutos. Modificar reglas de negocio existentes toma entre 1-2 horas dependiendo de la complejidad. Estos tiempos son consistentes a lo largo del proyecto, lo que sugiere que no estamos acumulando deuda técnica que ralentice el desarrollo.

V-D. Validación con usuarios y cumplimiento de requerimientos

Más allá de las métricas técnicas, el verdadero éxito del sistema se mide en qué tan bien satisface las necesidades de sus usuarios finales. Realizamos múltiples sesiones de pruebas de usabilidad y recolectamos feedback estructurado que nos dio insights valiosos.

Cumplimiento de requerimientos funcionales: Todas las 47 historias de usuario definidas inicialmente fueron implementadas y validadas con stakeholders de Electrohuila. Durante las sesiones de validación, que se realizaron en tres iteraciones a lo largo de dos meses, cada historia fue demostrada y validada individualmente.

Lo interesante fue que durante estas validaciones, los stakeholders solicitaron 8 cambios menores y 3 nuevas funcionalidades que no habían sido identificadas en el análisis inicial. Por ejemplo, no habíamos considerado la necesidad de ver un historial de todas las modificaciones hechas a una cita, pero resultó ser importante para auditoría. Pudimos implementar estas adiciones sin mayor dificultad gracias a la flexibilidad de la arquitectura.

Un indicador objetivo del cumplimiento funcional fue que en la última sesión de validación, los stakeholders aprobaron el 100 % de las funcionalidades sin solicitar cambios adicionales. El product owner de Electrohuila comentó que este era el primer proyecto de software que desarrollaban donde no había un gap significativo entre lo solicitado y lo entregado.

Evaluación de seguridad: Implementamos autenticación robusta usando JWT con refresh tokens, autorización basada en roles con permisos granulares, almacenamiento seguro de contraseñas usando bcrypt con factor de trabajo 12, y protecciones contra vulnerabilidades comunes (SQL injection, XSS, CSRF).

Para validar la seguridad del sistema, realizamos pruebas de penetración básicas usando herramientas como OWASP ZAP. La herramienta identificó 3 vulnerabilidades potenciales de severidad media relacionadas con headers de seguridad faltantes, que corregimos inmediatamente. No se identificaron vulnerabilidades de alta severidad.

Ejecutamos también pruebas manuales de seguridad intentando bypass de autenticación, inyección SQL, XSS, y CSRF. El sistema resistió exitosamente todos los ataques intentados. Particularmente, las inyecciones SQL fueron completamente neutralizadas por el uso de Entity Framework y stored procedures parametrizados. No logramos ejecutar ni una sola inyección SQL exitosa de las 47 que intentamos basadas en patrones conocidos.

Las contraseñas se almacenan usando bcrypt con un factor de trabajo de 12, lo que significa aproximadamente 150ms para hashear cada contraseña. Esto es deliberadamente lento para dificultar ataques de fuerza bruta, pero lo suficientemente rápido para no afectar la experiencia de usuario durante login.

Pruebas de usabilidad y satisfacción de usuarios: Realizamos sesiones de pruebas de usabilidad con 20 usuarios representativos: 12 ciudadanos de diferentes grupos demográficos y 8 empleados administrativos de Electrohuila. Las sesiones se realizaron en un laboratorio de usabilidad donde podíamos observar y grabar las interacciones.

Los resultados fueron muy alentadores. En el System Usability Scale (SUS), un cuestionario estandarizado de 10 preguntas, el sistema obtuvo un puntaje promedio de 78.5 sobre 100. Según la escala de interpretación del SUS, puntajes por encima de 68 se consideran por encima del promedio, y por encima de 80 se consideran excelentes. Nuestro puntaje de 78.5 nos coloca firmemente en el rango "bueno a excelente".

Desglosando por tipo de usuario, los empleados administrativos dieron un puntaje promedio de 82, mientras que los ciudadanos dieron 76.5. La diferencia es lógica considerando que los empleados tienen más funcionalidades disponibles y pasaron más tiempo aprendiendo el sistema.

El tiempo promedio para agendar una cita fue de 2 minutos y 18 segundos, considerablemente mejor que nuestro objetivo de 3 minutos. El usuario más rápido completó el proceso en 1 minuto y 5 segundos, mientras que el más lento tomó 5 minutos y 30 segundos (era una persona mayor de 70 años con poca experiencia con smartphones, pero aun así completó la tarea exitosamente sin ayuda).

Durante las sesiones de usabilidad, identificamos 7 puntos de confusión donde múltiples usuarios dudaban o cometían errores. El más común fue en la selección de fecha, donde el 30 % de los usuarios inicialmente intentaba

hacer click en fechas no disponibles esperando ver por qué no estaban disponibles. Agregamos un tooltip explicativo que resolvió este problema en iteraciones posteriores.

Evaluación de rendimiento percibido: Más allá de las métricas objetivas de rendimiento, nos interesaba el rendimiento percibido por los usuarios. Les pedimos que calificaran la velocidad del sistema en una escala de 1 (muy lento) a 5 (muy rápido), y obtuvimos un promedio de 4.3.

Los comentarios cualitativos fueron positivos: responde inmediatamente", "no tengo que esperar", "más rápido que otras aplicaciones del gobierno que he usado". Solo 2 de 20 usuarios mencionaron haber notado lentitud, y en ambos casos fue durante la carga inicial de la aplicación móvil, algo que podemos optimizar con lazy loading.

Medimos también el time to interactive, es decir, cuánto tiempo pasa desde que el usuario abre la aplicación hasta que puede interactuar con ella. Para la aplicación móvil, el promedio fue de 2.1 segundos en WiFi y 4.8 segundos en 4G, tiempos que consideramos aceptables para una aplicación híbrida.

Disponibilidad y confiabilidad: El sistema está diseñado para alta disponibilidad mediante manejo robusto de errores, reconexión automática de conexiones WebSocket, y degradación elegante ante fallos de componentes. Durante un mes de pruebas, el sistema mantuvo un uptime del 99.7 %, con solo dos interrupciones: una de 15 minutos por un reinicio planificado del servidor, y otra de 10 minutos por un problema de red fuera de nuestro control.

Simulamos varios escenarios de fallo para probar la resiliencia del sistema. Cuando apagamos abruptamente el servidor de base de datos, la aplicación mostró mensajes de error apropiados a los usuarios y comenzó a reintentar la conexión automáticamente. Una vez restaurada la base de datos, el sistema se recuperó completamente en menos de 30 segundos sin requerir intervención manual.

Cuando simulamos pérdida de conexión de red en dispositivos móviles, la aplicación mostró claramente el estado offline y permitió a los usuarios consultar citas previamente cargadas. Al restaurar la conexión, la aplicación se sincronizó automáticamente y descargó cualquier notificación pendiente.

El sistema implementa circuit breakers para servicios externos, aunque actualmente no dependemos de servicios externos críticos. Esta infraestructura está lista para cuando en el futuro se integren servicios como envío de SMS o correos electrónicos.

Observaciones inesperadas y aprendizajes: Durante las pruebas emergieron varios hallazgos que no habíamos anticipado. Por ejemplo, descubrimos que aproximadamente el 18 % de los usuarios intentan agendar citas para el mismo día, algo que nuestro sistema inicialmente no permitía. Después de discutir con Electrohuila, agregamos la capacidad de agendar citas con tan solo 2 horas de anticipación, lo cual ha sido muy utilizado.

Otro hallazgo interesante fue el patrón de uso del sistema de notificaciones. Observamos que los usuarios no descartan las notificaciones inmediatamente; las dejan acumuladas como recordatorios. Esto nos llevó a implementar un límite de 50 notificaciones guardadas por usuario y un sistema de auto-limpieza de notificaciones antiguas.

Notamos también que los administradores raramente usan la función de reportes avanzados que habíamos implementado, prefiriendo exportar datos crudos a Excel para hacer sus propios análisis. Esto sugiere que en futuras versiones deberíamos enfocarnos más en capacidades de exportación que en reportes predefinidos.

V-E. Síntesis de resultados

En conjunto, el proyecto resultó en un sistema empresarial completo, funcional, escalable y mantenible que demuestra la aplicación exitosa de principios modernos de ingeniería de software. Los números objetivos respaldan esta afirmación: tiempos de respuesta por debajo de 500ms en el percentil 95, throughput de 500 req/s, cobertura de pruebas del 72 %, puntaje SUS de 78.5, y uptime del 99.7 %.

Más importante que los números es la validación de que el sistema satisface necesidades reales de Electrohuila y sus ciudadanos. El feedback positivo de usuarios, la ausencia de bugs críticos durante las pruebas, y la facilidad con que el equipo técnico de Electrohuila pudo entender y mantener el sistema nos dan confianza de que este proyecto cumple su objetivo educativo de demostrar cómo construir software empresarial de calidad.

Los desafíos que enfrentamos - desde la optimización de consultas Oracle hasta el diseño de interfaces intuitivas para usuarios no técnicos - fueron oportunidades de aprendizaje que enriquecieron la experiencia formativa. El hecho de que pudimos superar estos desafíos y entregar un sistema funcional valida tanto las tecnologías elegidas como la metodología de desarrollo empleada.

VI. Discusión

La experiencia de desarrollar el Sistema de Agendamiento de Citas PQR para Electrohuila nos proporcionó lecciones que van mucho más allá del simple ejercicio de escribir código. Mirando atrás, nos damos cuenta de que cada decisión técnica que tomamos llevaba consigo implicaciones que no siempre anticipamos completamente en el momento. Este proyecto nos enseñó tanto sobre ingeniería de software moderna como sobre la importancia de comprender el contexto humano y organizacional en el que operaría nuestra solución.

VI-A. Decisiones arquitectónicas y sus consecuencias reales

La elección de Clean Architecture como patrón arquitectónico principal fue probablemente la decisión más significativa del proyecto, y ciertamente la más debatida dentro del equipo durante las primeras semanas. Recuerdo claramente las discusiones iniciales donde algunos miembros del equipo argumentaban que estábamos "sobreingenieriando"la solución, especialmente considerando que

este era un proyecto académico. En retrospectiva, entiendo perfectamente ese escepticismo inicial.

Clean Architecture introdujo una complejidad estructural que, durante las primeras dos o tres semanas, parecía ralentizarnos considerablemente. La necesidad de crear interfaces para cada repositorio, definir entidades de dominio separadas de los modelos de base de datos, y mantener DTOs distintos para cada caso de uso generó código que, superficialmente, parecía redundante. Hubo momentos de frustración donde nos preguntábamos si realmente necesitábamos toda esta infraestructura para un sistema de agendamiento de citas.

Sin embargo, la apuesta por esta arquitectura demostró su valor de manera dramática alrededor de la mitad del proyecto, cuando Electrohuila solicitó cambios significativos en cómo se manejaban las notificaciones. Gracias a la separación de capas, pudimos modificar completamente la implementación del servicio de notificaciones sin tocar ni una línea de la lógica de negocio. El equipo backend pudo trabajar en la nueva implementación usando SignalR mientras el equipo frontend continuaba desarrollando interfaces consumiendo los mismos contratos. Esta experiencia nos mostró visceralmente el valor de la inversión de dependencias y la separación de responsabilidades.

Dicho esto, la curva de aprendizaje fue considerable y, en un contexto puramente comercial con presiones de tiempo extremas, podría haber sido problemática. Los primeros pull requests eran objeto de largas sesiones de revisión donde debatíamos si cierta lógica pertenecía a la capa de aplicación o dominio, o si un determinado objeto debería ser una entidad o un value object. Estos debates, aunque a veces parecían bizantinos, fueron fundamentales para desarrollar una comprensión compartida de los principios arquitectónicos.

Para equipos menos experimentados o proyectos genuinamente pequeños, arquitecturas más simples como una estructura de tres capas tradicional (presentación, negocio, datos) probablemente serían más apropiadas. La pregunta clave que aprendimos a hacernos no es "¿cuál es la mejor arquitectura?"sino "¿cuál es la arquitectura apropiada dado nuestro contexto, equipo y requisitos de evolución futura?"

La decisión de utilizar Oracle Database merece una discusión aparte. Inicialmente, varios miembros del equipo preferían PostgreSQL, principalmente por familiaridad y por la riqueza de recursos en línea. La decisión de usar Oracle vino determinada por los requerimientos de Electrohuila, que opera toda su infraestructura de bases de datos sobre Oracle y tiene políticas corporativas estrictas al respecto.

En la práctica, esta decisión nos enseñó lecciones valiosas sobre trabajar con restricciones del mundo real. Si bien Entity Framework Core teóricamente abstrae las diferencias entre proveedores de bases de datos, descubrimos que "teoría≥ "práctica"no siempre coinciden perfectamente. El provider de Oracle tiene comportamientos específicos que requirieron atención especial. Por ejemplo, el mapeo automático de tipos .NET a tipos Oracle a veces producía resultados inesperados: un 'DateTime' en C# podría mapearse a 'DATE' o 'TIMESTAMP' dependiendo del contexto, y las diferencias semánticas entre estos tipos Oracle causaron bugs sutiles relacionados con zonas horarias.

Más frustrante aún fue el manejo de secuencias para generación de IDs. Oracle usa secuencias para autoincrementos en lugar del patrón 'IDENTITY' de SQL Server, y configurar Entity Framework para trabajar correctamente con secuencias requirió configuración explícita en Fluent API. Pasamos casi una tarde completa depurando un error donde los IDs generados se desincronizaban entre la aplicación y la base de datos porque la caché de secuencias de Oracle estaba configurada incorrectamente.

Estas experiencias nos enseñaron que la abstracción siempre tiene límites, y que conocer los detalles específicos de tu base de datos sigue siendo importante incluso cuando usas un ORM moderno. En hindsight, deberíamos haber dedicado más tiempo al inicio del proyecto a estudiar las particularidades del provider de Oracle en lugar de asumir que "simplemente funcionaría".

VI-B. El stack tecnológico: beneficios reales y dolores inesperados

La decisión de usar .NET para backend y móvil, combinado con Next.js para web, surgió de un análisis de trade-offs que ahora, viéndolo en retrospectiva, fue más acertado de lo que esperábamos en algunos aspectos y más problemático en otros.

La reutilización de código entre backend y móvil fue genuinamente transformadora. Inicialmente pensábamos que el beneficio principal sería compartir modelos de datos, pero el valor real resultó ser mucho más profundo. Compartimos validaciones completas, lógica de formateo de datos, y hasta pequeñas utilidades que garantizaron comportamiento absolutamente idéntico entre las plataformas. Hubo un momento particular que recuerdo claramente: habíamos implementado validación compleja para números de documento de identidad que debía verificar dígitos de control. Esta validación existía en el backend como parte de las reglas de negocio, pero también necesitábamos ejecutarla en el móvil para feedback inmediato al usuario. Gracias a usar C# en ambos lados, simplemente creamos un proyecto de biblioteca compartida y referenciamos la misma implementación exacta en ambos contextos. Esto no solo nos ahorró tiempo, sino que eliminó por completo la posibilidad de inconsistencias en esta validación crítica.

La seguridad de tipos end-to-end que obtuvimos usando TypeScript en frontend y C# en backend fue otro beneficio que apreciamos más con el tiempo. Durante las primeras iteraciones, algunos miembros del equipo sentían que los tipos eran "burocracia adicional"que ralentizaba el desarrollo. Pero cuando llegó el momento de refactorizar la estructura de DTOs para acomodar nuevos

requisitos, la diferencia fue noche y día comparado con proyectos anteriores en JavaScript puro. El compilador identificó automáticamente cada ubicación donde necesitábamos ajustar código para acomodar los cambios. En JavaScript tradicional, estos problemas habrían aparecido como errores en tiempo de ejecución, posiblemente solo en producción.

Sin embargo, no todo fue color de rosa con nuestro stack tecnológico. El ecosistema de MAUI, siendo relativamente joven, nos presentó desafíos que no anticipamos completamente. Mientras React Native o Flutter tienen bibliotecas maduras para casi cualquier funcionalidad que puedas imaginar, con MAUI frecuentemente tuvimos que implementar cosas desde cero o adaptar soluciones de Xamarin.Forms con resultados variables. Específicamente, encontrar un buen calendario/date picker que funcionara consistentemente en Android e iOS nos tomó más tiempo del que nos gustaría admitir. Terminamos probando tres bibliotecas diferentes antes de decidir implementar nuestra propia solución usando controles nativos.

La compatibilidad entre versiones de paquetes NuGet fue otra fuente constante de fricciones pequeñas pero acumulativas. Hubo una actualización particularmente problemática donde Entity Framework Core 9.0.1 tenía un conflicto sutil con Oracle.EntityFrameworkCore 9.21.121. El error solo aparecía en ciertas consultas complejas con múltiples JOINs, lo que hizo que la depuración fuera especialmente difícil. Pasamos dos días rastreando el problema antes de descubrir que había un issue abierto en GitHub y que la solución temporal era fijar una versión específica del provider de Oracle. Estas experiencias nos enseñaron la importancia de tener una estrategia clara de gestión de dependencias y de no actualizar indiscriminadamente.

Next.js 15 con Server Components fue quizás la parte más innovadora de nuestro stack, pero también la que presentó la curva de aprendizaje más empinada. El modelo mental de separar Server Components y Client Components es fundamentalmente diferente de cómo funcionaba React tradicionalmente, y al principio cometimos muchos errores. Los errores de hidratación eran particularmente frustrantes porque los mensajes de error a menudo no eran muy descriptivos. Recuerdo pasar una mañana completa depurando un error de hidratación que resultó ser causado por usar un hook ‘useEffect‘ en lo que pensábamos era un Server Component pero que inadvertidamente se había convertido en Client Component por tener una directiva ‘’use client’‘ heredada de un componente padre.

La solución no fue técnica sino de proceso: creamos convenciones claras de nomenclatura donde los Client Components tenían ‘.client.tsx‘ en su nombre de archivo, aunque Next.js no lo requiere. Esta convención simple hizo que fuera inmediatamente obvio al leer código qué componente era qué, y redujo dramáticamente estos errores.

VI-C. Desafíos técnicos específicos y lo que nos enseñaron

Algunos desafíos técnicos que enfrentamos merecen discusión detallada porque las lecciones que aprendimos de ellos fueron particularmente valiosas.

La saga de SignalR y las reconexiones: Implementar SignalR para notificaciones en tiempo real parecía straightforward según la documentación. La realidad fue considerablemente más complicada. El escenario básico funciona perfectamente: estableces conexión, envías mensajes, recibes mensajes. Pero los casos edge donde las cosas se vuelven interesantes son inevitables en producción.

El problema específico que encontramos fue este: cuando un usuario perdía temporalmente conectividad de red (algo común en móviles), SignalR intentaba reconectar automáticamente, lo cual es bueno. Pero si la desconexión duraba más de unos segundos, el servidor limpiaba los grupos a los que estaba suscrito el usuario. Cuando el cliente reconectaba exitosamente, no recibía notificaciones porque ya no estaba en los grupos apropiados. Esto resultó en un bug sutil donde las notificaciones ”simplemente no llegaban”de manera intermitente e impredecible.

La solución requirió implementar un protocolo de reconexión en el cliente que, después de reconectar, explícitamente se resuscribiera a todos los grupos relevantes. También implementamos backoff exponencial para reintentos, empezando con delays cortos y aumentándolos gradualmente para evitar bombardear el servidor. Esta lógica terminó siendo sorprendentemente compleja, con manejo de estados como çonectando”, reconectando”, ”desconectado-temporalmente”, y ”desconectado-permanentemente”.

Cuando desplegamos a múltiples instancias del backend (requerimiento de Electrohuila para alta disponibilidad), descubrimos otro problema: las notificaciones solo llegaban a usuarios conectados a la misma instancia del servidor que generó la notificación. Esto requirió configurar un backplane con Redis para sincronizar mensajes entre instancias. La configuración de Redis fue relativamente simple, pero depurar problemas de conectividad entre instancias en el entorno de staging de Electrohuila fue complejo debido a políticas de firewall y configuraciones de red que inicialmente no comprendíamos completamente.

En retrospectiva, deberíamos haber considerado estas complejidades desde el diseño inicial. Las notificaciones en tiempo real parecen una funcionalidad simple superficialmente, pero garantizar confiabilidad en escenarios de red impredecibles requiere pensamiento arquitectónico cuidadoso. Si tuviéramos que hacerlo de nuevo, dedicaríamos tiempo específicamente a modelar y probar escenarios de fallo desde el principio.

Server Components vs Client Components - más que una distinción técnica: Como mencioné antes, la diferencia entre Server Components y Client Components en Next.js 15 fue inicialmente confusa. Pero más allá de la confusión técnica, lo que resultó interesante fue cómo esta distinción nos forzó a pensar más cuidadosamente sobre la arquitectura del frontend.

Tradicionalmente en React, hay una tendencia a hacer todo interactivo "por si acaso". Un botón podría tener un hover effect, entonces le agregas estado. Un componente podría necesitar reaccionar a cambios futuros, entonces lo haces reactivo desde el inicio. Con Server Components, esta aproximación no funciona porque Server Components no pueden usar hooks ni estado.

Esto nos forzó a preguntarnos explícitamente para cada componente: "¿realmente necesita interactividad, o es puramente presentacional?.ªsí es como se hacen las cosas"Esta pregunta simple resultó en una arquitectura de frontend más limpia y deliberada. Componentes que genuinamente solo renderizan datos se mantuvieron como Server Components, lo que mejoró el performance porque menos JavaScript se enviaba al cliente. Componentes interactivos eran explícitamente Client Components con responsabilidades claras.

Un caso específico que ilustra esto: teníamos una tabla que mostraba citas agendadas. Inicialmente la hicimos completamente como Client Component porque .ºbviamente"necesitaría sorting, filtering, etc. Pero cuando implementamos estas funcionalidades, nos dimos cuenta de que para nuestro caso de uso (vistas individuales de usuarios con pocas citas), podíamos hacer el sorting y filtering en el servidor. Refactorizamos la tabla a Server Component, moviendo la lógica de filtrado a query parameters de URL. El resultado fue una tabla más rápida de cargar, con mejor SEO, y que mantenía estado de filtrado en la URL (permitiendo compartir links a vistas filtradas específicas).

MAUI data binding y las sutilezas de MVVM: El patrón MVVM en MAUI fue otro área donde la realidad práctica resultó más compleja que la teoría. El concepto es elegante: separas la vista (XAML) del modelo de vista (C#) usando data binding, y el framework se encarga de mantener sincronizado la UI con los datos.

En la práctica, el binding tiene reglas específicas que, si no se siguen exactamente, resultan en comportamiento silenciosamente incorrecto. Un problema común que enfrentamos repetidamente: una propiedad en el ViewModel cambia, pero la UI no se actualiza. El culpable casi siempre era olvidar implementar 'INotifyPropertyChanged' correctamente o no invocar 'OnPropertyChanged()' después de modificar un valor.

Esto se volvió especialmente problemático en propiedades calculadas. Por ejemplo, teníamos una propiedad 'CanBookAppointment' que dependía de múltiples otras propiedades ('IsLoggedIn', 'SelectedDate', 'SelectedTime', etc.). Cada vez que cualquiera de estas propiedades cambiaba, necesitábamos manualmente invocar 'OnPropertyChanged(nameof(CanBookAppointment))'. Olvidar una de estas invocaciones resultaba en botones que permanecían deshabilitados cuando deberían estar habilitados, o viceversa.

La solución llegó cuando descubrimos CommunityToolkit.MVVM con sus source generators. En lugar de escribir manualmente todo el boilerplate de 'INotifyPropertyChanged', podías decorar propiedades con atributos como '[ObservableProperty]' y el generator creaba automáticamente todo el código necesario. Para propiedades calculadas, '[NotifyPropertyChangedFor]' permitía declarativamente especificar dependencias.

Esta experiencia nos enseñó una lección más amplia: cuando un framework requiere mucho código boilerplate repetitivo, probablemente existe una herramienta o biblioteca que lo automatiza. No asumas que .ªsí es como se hacen las cosas"solo porque la documentación oficial muestra código manual. La comunidad a menudo desarrolla mejores patrones y herramientas.

Performance de Oracle y el arte de la optimización de queries: Un área donde nuestra inexperiencia inicial fue evidente fue en la optimización de consultas a base de datos. Los primeros queries que escribimos funcionaban correctamente pero, cuando probamos con volúmenes de datos realistas, algunos eran inaceptablemente lentos.

Un caso específico: teníamos una consulta que obtenía todas las citas de un usuario junto con información del servicio solicitado, el funcionario asignado, y el estado de la cita. La implementación inicial en LINQ era limpia y legible:

"'csharp var appointments = await $_{c}ontext.Appointments.Include(a => a.Service).Include(a => a.Employee).Include(a => a.Status).Where(a => a.UserId == userId).ToListAsync();$ '''

Con 10 citas en base de datos de desarrollo, esta consulta era instantánea. Con 50,000 citas en el ambiente de staging de Electrohuila, tomaba varios segundos. El problema era múltiple: primero, no teníamos índices apropiados en las columnas de foreign keys. Segundo, Entity Framework generaba queries con JOINs que Oracle no optimizaba bien. Tercero, estábamos cargando datos innecesarios (por ejemplo, todas las columnas de Employee cuando solo necesitábamos nombre y ID).

La optimización fue un proceso educativo. Primero, aprendimos a usar 'EXPLAIN PLAN' en Oracle para entender exactamente qué estaba haciendo la base de datos. Descubrimos que estaba haciendo table scans completos en lugar de usar índices. Agregamos índices en las foreign keys y columnas frecuentemente filtradas.

Segundo, refactorizamos el query para seleccionar solo columnas necesarias usando projections:

"'csharp var appointments = await $_{c}ontext.Appointments.Where(a => a.UserId == userId).Select(a => newAppointmentDtoId = a.Id, DateTime = a.DateTime, ServiceN$

Esto redujo el tiempo de ejecución dramáticamente porque transmitíamos menos datos sobre la red y Oracle podía optimizar mejor el plan de ejecución.

Para queries particularmente complejos con lógica de negocio significativa, terminamos creando stored procedures en Oracle. Esto fue controversial en el equipo - algunos argumentaban que movíamos lógica de negocio a la base

de datos, violando la arquitectura limpia. Otros argumentaban que queries complejos son una responsabilidad legítima de la capa de datos.

El compromiso al que llegamos fue este: la lógica de negocio permanece en la capa de aplicación, pero agregación y transformación de datos complejas puede hacerse en stored procedures para performance. Los stored procedures se invocan a través de interfaces de repositorio, manteniendo la inversión de dependencias. Esta decisión pragmática nos enseñó que los principios arquitectónicos son guías, no leyes absolutas, y el contexto importa.

VI-D. Lecciones aprendidas - más allá de lo técnico

Las lecciones más valiosas que aprendimos a menudo trascendieron lo puramente técnico y tocaron aspectos de proceso, comunicación y trabajo en equipo.

El diseño inicial realmente importa, pero no de la manera que pensábamos: Al inicio del proyecto, dedicamos aproximadamente tres semanas a diseño antes de escribir código de producción. Esto incluyó diseño de base de datos, definición de arquitectura, creación de diagramas de componentes, y desarrollo de mockups de UI. Algunos miembros del equipo sentían que estábamos "perdiendo tiempoz querían .empezar a programar de verdad".

En retrospectiva, esas tres semanas fueron probablemente las más productivas del proyecto. Pero no exactamente por las razones que esperábamos. El diagrama de base de datos que creamos cuidadosamente terminó siendo modificado significativamente durante el desarrollo - lo cual está bien. Los mockups de UI que diseñamos cambiaron después de feedback de usuarios - también está bien.

El valor real del diseño inicial no fue producir un plan perfecto que luego ejecutamos mecánicamente. El valor fue crear un entendimiento compartido entre el equipo. Las discusiones durante las sesiones de diseño alinearon nuestras expectativas sobre qué estábamos construyendo y por qué. Cuando surgían preguntas durante la implementación, frecuentemente podíamos resolverlas recordando el razonamiento detrás de decisiones de diseño.

Si tuviéramos que hacerlo de nuevo, seríamos más explícitos sobre este propósito. En lugar de tratar el diseño como çrear documentación", lo trataríamos como "desarrollar entendimiento compartido". La documentación es un subproducto útil, pero secundario.

Testing temprano vs testing tardío - una diferencia de filosofía: Una decisión consciente que tomamos fue implementar pruebas unitarias desde el primer momento, no dejarlas para "después cuando tengamos tiempo". Esta decisión no fue unánime - hubo resistencia de algunos miembros que argumentaban que escribir tests ralentizaría el desarrollo inicial.

Los primeros días, parecían tener razón. Escribir un caso de uso simple con sus respectivas pruebas tomaba el doble de tiempo que solo escribir el código. Pero alrededor de la cuarta o quinta semana del proyecto, algo cambió. Empezamos a modificar código existente para nuevos requisitos. Aquí, las pruebas transformaron completamente la experiencia.

Con pruebas, podías refactorizar código confidentemente. Si las pruebas pasaban después de tus cambios, tenías alta confianza de que no habías roto funcionalidad existente. Sin pruebas, cada cambio requería testing manual extensivo, o peor, resultaba en regresiones que solo se descubrían más tarde.

Un momento específico que solidificó el valor de testing para todo el equipo: estábamos modificando la lógica de validación de citas para acomodar nuevos requisitos de negocio. Después de hacer los cambios, ejecutamos las pruebas y 15 tests fallaron. Inicialmente esto pareció desalentador, pero resultó ser invaluable. Cada test que falló identificaba un caso específico que nuestra nueva implementación no manejaba correctamente. Pudimos sistemáticamente revisar cada falla, corregir el código, y verificar que la corrección funcionaba. Sin estas pruebas, cada uno de esos 15 casos habría sido un bug potencial en producción.

Documentación continua vs documentación final: Adoptamos una política de documentar decisiones importantes continuamente usando Architecture Decision Records (ADRs). Cada vez que tomábamos una decisión arquitectónica significativa, creábamos un documento breve describiendo el contexto, la decisión, y el razonamiento.

Inicialmente esto pareció overhead adicional. Pero el valor se manifestó rápidamente cuando nuevos miembros se unieron al proyecto a mitad del camino. En lugar de explicar repetidamente por qué habíamos tomado ciertas decisiones, podíamos simplemente señalar a los ADRs relevantes. Más aún, nosotros mismos consultábamos estos documentos cuando necesitábamos recordar por qué habíamos hecho algo de cierta manera.

Los comentarios en código fueron otra área donde el enfoque continuo superó documentar al final. Código complejo que parecía obvio mientras lo escribías frecuentemente resultaba misterioso algunas semanas después. Comentarios explicando el "por qué"(no el "qué", que debería ser obvio del código mismo) fueron invaluables para mantenimiento futuro.

El feedback de usuarios reales es insustituible: Esto parece obvio, pero la diferencia entre lo que creíamos que querían los usuarios y lo que realmente necesitaban fue frecuentemente sorprendente. Organizamos sesiones de validación con personal de Electrohuila aproximadamente cada dos semanas donde demostrábamos funcionalidad y recibíamos feedback.

Un ejemplo específico: habíamos diseñado el flujo de agendamiento de citas con un proceso de múltiples pasos: primero seleccionas el servicio, luego la fecha, luego la hora, luego confirmas. Nos parecía lógico y bien estructurado. Los usuarios reales encontraron esto tedioso y preferían ver disponibilidad de múltiples días a la vez en un calendario,

luego seleccionar un slot específico. Su modelo mental era fundamentalmente diferente al nuestro.

Incorporar este feedback requirió refactorizar significativamente la UI de agendamiento, pero el resultado fue un sistema mucho más usable. Esta experiencia nos enseñó humildad sobre nuestras propias suposiciones y el valor incalculable de validación continua con usuarios reales.

Gestión de dependencias - entre estabilidad e innovación: Nuestra política inicial era mantenernos en versiones estables de todas las dependencias y actualizar conservadoramente. Esto funcionó bien para evitar problemas de compatibilidad, pero significaba que a veces nos perdíamos mejoras importantes.

El punto de inflexión fue cuando Next.js 15 introdujo Server Components. Estábamos en Next.js 14 y la actualización parecía arriesgada a mitad del proyecto. Decidimos hacerla de todos modos, y aunque causó algunos días de trabajo ajustando código, las mejoras de performance y desarrollador experience valieron la pena.

La lección fue que no existe una política de actualización universalmente correcta. Dependencias críticas de infraestructura (base de datos, runtime) deberían ser muy conservadoras. Frameworks y bibliotecas donde las mejoras proporcionan valor directo pueden justificar actualizaciones más agresivas. El contexto y el análisis de riesgo/beneficio son esenciales.

VI-E. Aplicabilidad más allá de este proyecto específico

Reflexionando sobre la transferibilidad de este trabajo, vemos aplicabilidad en múltiples dimensiones.

Las decisiones técnicas - Clean Architecture, el stack tecnológico, los patrones de diseño - son ciertamente transferibles a otros proyectos de sistemas de agendamiento o gestión de citas en el sector público. Pero lo que podría ser más valioso son las lecciones de proceso: la importancia de diseño colaborativo inicial, testing temprano, documentación continua, y validación frecuente con usuarios.

Para instituciones educativas como el SENA, este proyecto demuestra que es absolutamente posible desarrollar aplicaciones empresariales de calidad en contextos de formación. Pero requiere varias condiciones: primero, requerimientos de negocio reales y claros, no ejercicios artificiales. Segundo, aplicación rigurosa de principios de ingeniería de software, no solo "programar hasta que funcione". Tercero, tiempo suficiente para iterar y aprender de errores - este no fue un proyecto de seis semanas, sino de varios meses.

La colaboración con Electrohuila fue fundamental para el éxito educativo del proyecto. Tener stakeholders reales con necesidades genuinas proporcionó motivación y contexto que proyectos puramente académicos no pueden replicar. Los aprendices entendieron que su trabajo tendría impacto real en ciudadanos reales, lo cual elevó significativamente el nivel de compromiso y calidad.

En hindsight, si tuviéramos que replicar este modelo de proyecto educativo-empresarial, enfatizaríamos aún más la comunicación continua con stakeholders. Inicialmente teníamos reuniones cada dos semanas; en retrospectiva, sesiones semanales más cortas habrían sido más efectivas. Feedback más frecuente habría prevenido algunos caminos equivocados donde desarrollamos funcionalidad que luego requirió cambios significativos.

También invertiríamos más tiempo en capacitación específica en tecnologías nuevas como MAUI y Next.js Server Components antes de comenzar desarrollo intensivo. Aprender mientras construyes es inevitable y valioso, pero cierta base teórica sólida habría reducido el tiempo gastado en errores triviales por incomprensión de conceptos fundamentales.

Finalmente, un aspecto que subestimamos fue la importancia de configurar ambientes de desarrollo consistentes desde el inicio. Gastamos tiempo significativo depurando problemas que resultaban ser diferencias entre ambientes de desarrollo locales. La adopción temprana de contenedores Docker para servicios compartidos (base de datos, Redis) habría sido valiosa. Implementamos esto eventualmente, pero hacerlo desde el día uno habría prevenido frustraciones.

En conclusión, este proyecto fue tanto un éxito técnico como educativo, pero quizás más importante, fue una experiencia de aprendizaje humilde. Aprendimos que el desarrollo de software no es principalmente sobre código, sino sobre personas: entender usuarios, colaborar en equipos, comunicarse con stakeholders, y aprender continuamente de experiencias. Las tecnologías y herramientas cambian constantemente, pero estas lecciones fundamentales sobre el proceso humano de crear software permanecen relevantes.

VII. Conclusiones y trabajo futuro

El desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila ha representado un viaje técnico y educativo significativo, demostrando que la aplicación rigurosa de principios modernos de ingeniería de software puede producir resultados tangibles incluso en contextos de formación práctica. Más allá de cumplir con los requerimientos funcionales establecidos, este proyecto ha servido como laboratorio para explorar las tensiones inherentes entre teoría arquitectural y restricciones pragmáticas, entre aspiraciones de calidad técnica y presiones de tiempo del mundo real.

La implementación de este sistema completo—abarcando aplicación móvil multiplataforma, portal web administrativo, API RESTful robusta, y notificaciones en tiempo real—no estuvo exenta de desafíos. Cada decisión técnica implicó compromisos: elegir Clean Architecture significó aceptar mayor complejidad inicial a cambio de mantenibilidad futura; adoptar .NET MAUI implicó navegar documentación aún en maduración; integrar múltiples tecnologías front-end requirió armonizar ecosistemas con filosofías distintas. Sin embargo, es precisamente en estas tensiones donde emergieron las lecciones más valiosas.

VII-A. Contribuciones principales

Este trabajo contribuye al campo de los sistemas de información empresariales de maneras que trascienden la simple documentación técnica. Las contribuciones se sitúan en la intersección entre teoría arquitectural, práctica de desarrollo, y contexto educativo:

1. **Documentación exhaustiva de arquitectura full-stack moderna integrada:** Mientras que la literatura técnica abunda en documentación de tecnologías individuales—.NET Core, Next.js, .NET MAUI, Oracle Database—existe una escasez notable de casos documentados que integren estas tecnologías en un sistema cohesivo funcionando en conjunto. Esta investigación no solo documenta la arquitectura final, sino el proceso iterativo de integración, incluyendo los puntos de fricción encontrados y sus resoluciones.
   Por ejemplo, la integración entre SignalR (en el backend .NET) y clientes Next.js requirió configuraciones específicas de CORS, manejo de headers personalizados, y estrategias de reconexión que no están completamente documentadas en fuentes oficiales. De manera similar, la comunicación entre la aplicación .NET MAUI y la API requirió consideraciones especiales de serialización, manejo de tokens, y gestión de conectividad intermitente en dispositivos móviles. Estas experiencias prácticas, documentadas con detalle, pueden servir como referencia valiosa para equipos enfrentando integraciones similares.
2. **Estudio aplicado de Clean Architecture en contexto empresarial real con restricciones concretas:** La literatura sobre Clean Architecture, Domain-Driven Design, y arquitecturas limpias generalmente presenta estos conceptos en escenarios idealizados o ejemplos simplificados. Este proyecto documenta la aplicación de Clean Architecture en un contexto con restricciones reales: plazos definidos, equipo en formación, base de datos legacy preexistente (Oracle), y requerimientos cambiantes durante el desarrollo.
   Se documentaron no solo los beneficios teóricos sino los impactos medibles: reducción del 60 % en tiempo de incorporación de nuevas funcionalidades una vez establecida la arquitectura base, disminución significativa en regresiones durante cambios por la separación clara de responsabilidades, y facilidad para implementar testing automatizado por la independencia de capas. Crucialmente, también se documentaron los costos: aproximadamente 30 % más tiempo invertido en etapas iniciales para estructurar capas apropiadamente, curva de aprendizaje pronunciada para el equipo, y tentación constante de çortar camino"que requirió disciplina para mantener.
3. **Implementación completa de comunicación bidireccional en tiempo real con consideraciones de producción:** La documentación existente de SignalR tiende a enfocarse en escenarios de demostración: salas de chat simples, actualizaciones de dashboard básicas. Este proyecto implementó un sistema de notificaciones empresarial completo, abordando desafíos de escalabilidad (uso de backplane de Redis para múltiples servidores), resiliencia (reconexión automática con exponential backoff), seguridad (autenticación de conexiones, autorización por grupos), y experiencia de usuario (sincronización de estado, manejo de notificaciones perdidas durante desconexión).
   La experiencia reveló patrones útiles: la importancia de implementar heartbeats para detectar conexiones zombies, la necesidad de almacenamiento persistente de notificaciones críticas (no solo broadcasting en tiempo real), y estrategias para degradar gracefully cuando la conexión en tiempo real no está disponible. Estas lecciones prácticas complementan la documentación técnica oficial con consideraciones de implementación del mundo real.
4. **Validación de educación técnica práctica produciendo software de calidad empresarial:** Existe un debate persistente sobre si la formación técnica práctica (como la del SENA en Colombia) puede producir profesionales capaces de desarrollar software que cumpla estándares empresariales rigurosos. Este proyecto proporciona evidencia de que, con mentoría apropiada, aplicación disciplinada de metodologías, y énfasis en principios fundamentales por encima de recetas, es absolutamente posible.
   El código producido no solo funciona sino que pasa revisiones de calidad estrictas, sigue convenciones establecidas de la industria, incluye documentación comprehensiva, y está estructurado para mantenibilidad a largo plazo. Este resultado sugiere que el factor determinante no es el tipo de institución educativa sino la rigurosidad del proceso y el compromiso con la excelencia técnica. La experiencia puede servir como modelo para otras instituciones educativas buscando elevar la calidad de proyectos formativos.
5. **Caso de estudio de modernización tecnológica en entidad pública:** Las empresas del sector público en Colombia frecuentemente enfrentan desafíos particulares: sistemas legacy complejos, procesos burocráticos establecidos, resistencia al cambio, y presupuestos limitados para innovación tecnológica. Este proyecto documenta un camino viable para modernización incremental: comenzar con un módulo específico (agendamiento de citas), implementarlo con tecnologías modernas, y establecer una base arquitectural que permita expansión futura.
   La estrategia de no intentar reemplazar todos los sistemas simultáneamente, sino proporcionar valor inmediato con un subsistema bien delimitado, demostró ser efectiva. Este enfoque pragmático de modernización puede ser replicable en otras entidades gubernamentales enfrentando desafíos similares.

VII-B. Cumplimiento de objetivos y reflexiones sobre resultados

El sistema desarrollado no solo cumple formalmente con los objetivos establecidos al inicio del proyecto, sino que en varios aspectos superó las expectativas iniciales. Sin embargo, el verdadero aprendizaje provino tanto de los éxitos como de los obstáculos encontrados en el camino:

- **Modernización del proceso de agendamiento:** El reemplazo de procesos manuales fragmentados con una solución digital unificada cumplió plenamente el objetivo técnico. Sin embargo, la reflexión importante aquí es que la modernización tecnológica va más allá del código: implica cambio organizacional, capacitación de usuarios, y ajustes a procesos establecidos. El sistema técnicamente superior solo genera valor cuando es adoptado y utilizado efectivamente por las personas. Esta lección sobre la importancia del factor humano en proyectos tecnológicos es quizás tan valiosa como las lecciones técnicas.
- **Mejora de la experiencia ciudadana:** La aplicación móvil intuitiva efectivamente permite a ciudadanos gestionar citas sin interacción telefónica o presencial. Las pruebas con usuarios reales revelaron algo interesante: incluso cuando la interfaz es técnicamente bien diseñada, algunos ciudadanos—particularmente adultos mayores o personas con poca familiaridad tecnológica—requieren acompañamiento inicial. Esto sugiere que el despliegue completo debería incluir campañas educativas, tutoriales integrados, y posiblemente asistencia inicial en puntos de atención presencial. La lección más profunda aquí es sobre diseño inclusivo: una aplicación no es verdaderamente accesible solo por tener interfaz limpia y navegación clara, sino cuando considera activamente las capacidades diversas de toda la población objetivo.
- **Eficiencia operacional para administradores:** El portal administrativo efectivamente centraliza gestión de empleados, sucursales, y configuraciones. Métricas preliminares sugieren reducción aproximada del 40 % en tiempo dedicado a tareas administrativas rutinarias comparado con el proceso anterior. Sin embargo, la interfaz administrativa reveló una tensión interesante: los usuarios administrativos frecuentemente prefieren densidad de información (ver mucha información simultáneamente) sobre minimalismo visual, contrario a usuarios finales que prefieren interfaces más simples y guiadas.
  Esto ilustra que no existe una "mejor"filosofía de diseño universal, sino que el diseño apropiado depende críticamente del contexto, usuarios, y objetivos específicos de cada interfaz.
- **Sistema de notificaciones en tiempo real funcionando robustamente:** El sistema de notificaciones cumple su objetivo de mantener informados a todos los actores sobre cambios relevantes. La implementación de SignalR con backplane de Redis permite escalar horizontalmente, y el manejo de reconexiones automáticas proporciona experiencia fluida incluso con conectividad inestable.
  Una observación importante: durante pruebas con usuarios, se descubrió que demasiadas notificaciones pueden ser contraproducentes, causando fatiga y eventualmente ignorándose. Fue necesario implementar configuraciones granulares permitiendo a usuarios controlar qué notificaciones recibir. Esta lección sobre balancear información útil con sobrecarga cognitiva es aplicable más allá de este proyecto específico.
- **Arquitectura escalable y mantenible validada en práctica:** La estructura arquitectural implementada efectivamente facilitó incorporación de nuevas funcionalidades con impacto mínimo en código existente. Por ejemplo, añadir soporte para nuevos tipos de PQR requirió cambios localizados en capas específicas sin modificar infraestructura general. La separación clara de responsabilidades también facilitó testing: cada capa puede probarse independientemente con mocks apropiados.
  Sin embargo, la escalabilidad y mantenibilidad no son características binarias sino espectros continuos que requieren inversión y disciplina constantes. El desafío futuro será mantener esta calidad arquitectural a medida que el sistema evoluciona y nuevos desarrolladores se incorporan.

VII-C. Limitaciones, desafíos encontrados, y aprendizajes

Toda investigación y desarrollo tiene fronteras, y reconocer explícitamente las limitaciones es esencial para contexto apropiado y honestidad intelectual. Las limitaciones de este proyecto no representan fallas sino decisiones conscientes de alcance y áreas identificadas para evolución futura:

- **Integración limitada con sistemas existentes de Electrohuila:** El sistema funciona de manera autónoma, lo cual simplificó desarrollo inicial pero crea silos de información. Ciudadanos con cuentas en otros sistemas de Electrohuila deben crear credenciales nuevas; información de PQR históricas no está accesible; datos de facturación permanecen desconectados.
  Esta limitación fue decisión consciente de alcance: integrar completamente con sistemas legacy habría requerido acceso extensivo a bases de datos y APIs no documentadas, multiplicando complejidad y riesgos. El enfoque de çomenzar independiente con plan de integración futura.[es] pragmático, pero la deuda técnica de esta desconexión acumulará hasta que se aborde. La lección aquí es sobre balancear pureza técnica (integración total desde día uno) con pragmatismo (entregar valor incremental).
- **Capacidades de reportería y analytics en etapa inicial:** Los reportes implementados cubren necesi-

dades operacionales básicas: citas por período, utilización de sucursales, empleados más solicitados. Sin embargo, el sistema carece de analytics sofisticados: predicción de demanda, identificación de patrones estacionales, análisis de comportamiento de usuarios, detección de anomalías.

La ausencia de analytics avanzados significa que decisiones de negocio importantes—como asignación de personal, expansión de capacidad, optimización de horarios—aún dependen de intuición más que datos. Implementar analytics es complejo: requiere infraestructura de data warehousing, pipelines de ETL, herramientas de visualización, y expertise en ciencia de datos. Reconocemos esta como área crítica para trabajo futuro.

- **Testing en dispositivos físicos limitado a muestra pequeña:** Las pruebas móviles se realizaron primariamente en emuladores Android e iOS, complementados con pruebas en aproximadamente 10 dispositivos físicos de miembros del equipo y colaboradores cercanos. Esta muestra es insuficiente para garantizar funcionamiento correcto en la diversidad de dispositivos del mercado: diferentes tamaños de pantalla, versiones de sistema operativo, fabricantes con personalizaciones, capacidades de hardware variadas.

  La realidad es que testing exhaustivo en dispositivos requiere infraestructura (device farms), tiempo, y presupuesto más allá de lo disponible en este proyecto. Sin embargo, deployment masivo sin este testing representa riesgo: pueden existir bugs específicos a dispositivos o configuraciones no encontrados en testing limitado. La estrategia de mitigación propuesta es lanzamiento gradual (phased rollout) con monitoreo intensivo de crashes y errores, permitiendo identificar y corregir problemas antes de expansión total.
- **Sistema de métricas y observabilidad incompleto:** Aunque se implementó logging básico y manejo de errores, el sistema carece de observabilidad comprehensiva. No hay métricas detalladas de rendimiento en producción, trazabilidad distribuida de requests entre servicios, dashboards de salud del sistema, o alertas proactivas sobre degradación de servicio.

  Esta es limitación significativa que afectará operación en producción: cuando ocurran problemas—y ocurrirán—la capacidad de diagnosticar rápidamente se verá comprometida. Implementar observabilidad apropiada (usando herramientas como Application Insights, Prometheus, Grafana, o ELK stack) debería ser prioridad antes de lanzamiento a gran escala. La lección es que observabilidad no es "nice to have"sino componente esencial de sistemas productivos.
- **Consideraciones de seguridad avanzadas pendientes:** Mientras que se implementaron prácticas de seguridad fundamentales (autenticación, autorización basada en roles, validación de input, uso de HTTPS, protección contra CSRF), existen aspectos de seguridad más avanzados no completamente abordados: auditoría comprehensiva de acciones sensitivas, rate limiting para prevenir abuso, WAF (Web Application Firewall), penetration testing formal, plan de respuesta a incidentes de seguridad.

  En el contexto de entidad pública manejando datos ciudadanos, seguridad robusta no es opcional. Antes de deployment amplio, se recomienda auditoría de seguridad por profesionales especializados, remediación de vulnerabilidades encontradas, y establecimiento de políticas de seguridad claras.
- **Documentación de usuario final y materiales de capacitación básicos:** Mientras que el código está bien documentado técnicamente, la documentación para usuarios finales (manuales de usuario, videos tutoriales, FAQs) es mínima. De manera similar, materiales para capacitar personal administrativo son limitados.

  Experiencias previas sugieren que adopción exitosa de sistemas nuevos correlaciona fuertemente con calidad de capacitación y soporte. Invertir en documentación y materiales de capacitación de alta calidad debería ser parte integral de la estrategia de deployment, no un añadido posterior.

VII-D. Trabajo futuro: visión de evolución y expansión

El sistema actual representa una base sólida, pero las posibilidades de evolución son amplias. El trabajo futuro no es simplemente una lista de pendientes sino una visión de cómo el sistema puede crecer para generar valor progresivamente mayor:

VII-D1. Corto plazo (3-6 meses):

- **Integración bidireccional con sistema de gestión de clientes (CRM):** Implementar conectores que sincronicen información de clientes entre el sistema de agendamiento y el CRM corporativo de Electrohuila. Esto eliminaría necesidad de credenciales duplicadas, permitiría acceso a historial completo de interacciones del cliente, y enriquecería el perfil ciudadano con contexto adicional.

  Técnicamente, esto requeriría: análisis del esquema de base de datos del CRM legacy, desarrollo de capa de abstracción para mapear modelos de datos diferentes, implementación de sincronización periódica o en tiempo real (dependiendo de requerimientos de freshness), y manejo robusto de errores y conflictos de sincronización. El impacto esperado sería experiencia significativamente mejorada para ciudadanos (no más múltiples registros) y visión unificada para personal administrativo.
- **Implementación de módulo de analytics operacionales básicos:** Desarrollar dashboards que visualicen métricas clave: tendencias de agendamiento, tasas de asistencia/ausencia, tiempos promedio de

atención, distribución geográfica de usuarios, horas pico de demanda. Estas métricas permitirían decisiones operacionales informadas.

La implementación podría utilizar tecnologías como Power BI integrado o desarrollo custom con bibliotecas de visualización (Chart.js, D3.js). Los datos alimentarían proceso ETL nocturno hacia data warehouse dimensional, permitiendo consultas analíticas sin impactar base de datos operacional.

- **Mejoras de accesibilidad conformes a estándares WCAG 2.1 AA:** Implementar soporte completo para lectores de pantalla (roles ARIA apropiados, labels descriptivos), navegación completa por teclado, contraste de colores conforme a estándares, soporte para ampliación de texto sin pérdida de funcionalidad, y opciones de personalización visual (modo alto contraste, tamaños de fuente ajustables).

  Esto requeriría auditoría de accesibilidad usando herramientas automatizadas (aXe, WAVE) complementadas con testing manual por usuarios con necesidades de accesibilidad diversas. El impacto social sería significativo: hacer el sistema verdaderamente accesible para todos los ciudadanos independiente de capacidades.
- **Notificaciones push nativas para aplicación móvil:** Implementar notificaciones push usando Firebase Cloud Messaging (para Android) y Apple Push Notification Service (para iOS). Actualmente, notificaciones solo funcionan cuando aplicación está abierta (SignalR requiere conexión activa). Notificaciones push permitirían alertar usuarios sobre recordatorios de citas, cambios, o mensajes importantes incluso cuando app está cerrada.

  Esto mejoraría significativamente engagement y reduciría ausencias a citas por olvido.

VII-D2. Mediano plazo (6-12 meses):

- **Sistema de colas virtuales y gestión de espera inteligente:** Extender funcionalidad más allá de agendamiento puro hacia gestión completa de flujo ciudadano. Ciudadanos podrían recibir "ticket virtual.ªl llegar a sucursal, monitorear su posición en cola en tiempo real desde app, y recibir notificación cuando se acerque su turno. Esto permitiría a ciudadanos esperar en ubicaciones convenientes (cafeterías cercanas, vehículo) en lugar de salas de espera congestionadas.

  La implementación requeriría: sistema de generación de tickets, algoritmo de predicción de tiempos de espera basado en métricas históricas, interfaz para personal que "llame"siguiente ticket, y pantallas en sucursales mostrando números activos. El impacto en experiencia ciudadana sería transformacional, reduciendo percepción de tiempo de espera y mejorando satisfacción.
- **Motor de recomendaciones y optimización de asignaciones:** Implementar sistema que sugiera automáticamente mejor empleado, sucursal, y hora para cada ciudadano basándose en: tipo de PQR, ubicación geográfica del ciudadano, disponibilidad en tiempo real, expertise específico requerido, y preferencias históricas. Machine learning podría optimizar estas sugerencias continuamente.

  Esto requeriría: recolección de datos históricos suficientes (al menos 6 meses), desarrollo de features para modelo de ML, entrenamiento de modelo de recomendación, y UI para presentar sugerencias sin eliminar control del usuario. El beneficio sería utilización más eficiente de recursos y tiempos de resolución potencialmente menores.
- **Módulo de satisfacción y feedback continuo:** Implementar sistema que solicite feedback de ciudadanos después de cada interacción: calificación del servicio, comentarios sobre experiencia, identificación de puntos de fricción. Datos agregados alimentarían análisis de sentimiento y detección de problemas sistémicos.

  Crucialmente, este sistema debería cerrar el loop: feedback debería visibilizarse a personal relevante, generar alertas sobre problemas emergentes, e idealmente trigger acciones correctivas automáticas donde apropiado. Este ciclo de mejora continua basado en feedback real es esencial para evolución del sistema.
- **Integración con plataformas de identidad nacional (autenticación mejorada):** Integrar con servicios de identidad digital del gobierno colombiano (como Carpeta Ciudadana Digital cuando esté completamente operacional) para autenticación robusta usando cédula digital. Esto mejoraría seguridad, simplificaría registro (pre-población de datos personales desde fuentes oficiales), y abriría posibilidades de integración con otros servicios gubernamentales.

VII-D3. Largo plazo (12+ meses):

- **Expansión a ecosistema completo de servicios ciudadanos:** Evolucionar el sistema de agendamiento específico de PQR hacia plataforma completa de interacción ciudadana con Electrohuila. Esto podría incluir: consulta de facturación, reporte de averías, solicitud de nuevos servicios, seguimiento de proyectos de infraestructura, comunicación bidireccional con empresa.

  La visión es que ciudadanos tengan punto único de contacto digital con Electrohuila para todas sus necesidades, con experiencia consistente y datos unificados. Técnicamente, esto requeriría arquitectura de microservicios robusta, API gateway, gestión de identidad centralizada, y probable migración a cloud para manejar escala y variabilidad de demanda.
- **Implementación de analytics predictivos con machine learning:** Ir más allá de reportería descriptiva (qué pasó) hacia analytics predictivos (qué pasará) y prescriptivos (qué hacer). Modelos de ML

podrían predecir: demanda futura de citas por tipo y ubicación, probabilidad de ausencia de ciudadanos individuales, tiempos esperados de resolución de PQR, identificación temprana de escalamiento de problemas.

Estas capacidades permitirían planificación proactiva de recursos, intervenciones preventivas (por ejemplo, confirmación adicional para ciudadanos con alta probabilidad de ausencia), y optimización continua de operaciones. La implementación requeriría infraestructura de data science (Jupyter notebooks, entrenamiento de modelos, MLOps), expertise especializado, y calidad de datos rigurosa.

- **Apertura de APIs públicas y ecosistema de desarrolladores:** Considerar apertura de APIs públicas (con apropiadas consideraciones de seguridad y rate limiting) permitiendo que terceros desarrollen aplicaciones complementarias. Por ejemplo, desarrolladores podrían crear integraciones con calendarios personales, asistentes virtuales, o aplicaciones comunitarias.

  Esto requeriría documentación exhaustiva de APIs, portal de desarrolladores, programas de certificación, y posiblemente sandbox environments. El beneficio sería innovación crowd-sourced: ideas y casos de uso no anticipados por equipo interno podrían emerger de comunidad externa.
- **Investigación de interfaces conversacionales y asistentes virtuales:** Explorar integración de chatbots basados en NLP (procesamiento de lenguaje natural) o asistentes de voz que permitan a ciudadanos interactuar con sistema usando lenguaje natural. ”Quiero agendar cita para reclamo de facturación el próximo martes por la tarde cerca de mi casa”podría interpretarse y procesarse automáticamente.

  Tecnologías como GPT-4, Claude, o modelos especializados en español podrían alimentar estos asistentes. El desafío es balancear capacidad impresionante de modelos grandes con necesidad de respuestas precisas y confiables en contexto de servicio público. Interfaces conversacionales podrían particularmente beneficiar usuarios con baja alfabetización digital o preferencia por interacción más natural.

VII-E. Reflexiones finales: impacto, aprendizajes, y perspectiva

Más allá de líneas de código, arquitecturas, y funcionalidades implementadas, este proyecto ha sido fundamentalmente sobre resolver problemas reales para personas reales. El impacto verdadero no se mide en métricas técnicas sino en ciudadanos que pueden agendar citas convenientemente sin ausentarse del trabajo, en personal administrativo que puede enfocarse en tareas de valor agregado en lugar de gestión manual de agendas, y en una organización pública que toma pasos concretos hacia modernización digital.

El proceso de desarrollo reveló verdades frecuentemente oscurecidas en discusiones puramente técnicas: que la mejor arquitectura es inútil si usuarios no adoptan el sistema, que perfección técnica debe balancearse con restricciones de tiempo y recursos, que comunicación clara entre stakeholders es tan crítica como comunicación clara entre componentes de software, que decisiones técnicas tienen implicaciones organizacionales y humanas profundas.

Una reflexión personal importante: desarrollar software de calidad requiere disciplina consistente, no solo en momentos de inspiración sino en el trabajo rutinario diario. Seguir principios arquitecturales cuando la fecha límite se aproxima y el .ªtajo.es tentador. Escribir tests comprehensivos cuando parece que funcionamiento manual es suficiente. Documentar decisiones cuando el contexto parece obvio ahora pero será oscuro en seis meses. Refactorizar código funcional para mejorar claridad. Estas prácticas, a menudo vistas como .ºverhead.º ”perfectionism”, son precisamente lo que separa software profesional de scripts descartables.

Este proyecto también ilustra la importancia crítica de mentoría y transferencia de conocimiento en formación técnica. Los estudiantes y técnicos en formación no carecen de capacidad intelectual o motivación, sino frecuentemente de exposición a prácticas profesionales reales, estándares de calidad de la industria, y contexto de por qué ciertas prácticas importan. Con guía apropiada y expectativas claras, pueden producir trabajo que rivaliza con profesionales experimentados.

Mirando hacia el futuro, el verdadero éxito de este proyecto no será el sistema en su estado actual sino su capacidad de evolucionar. Software no es estático: requiere mantenimiento continuo, adaptación a requerimientos cambiantes, incorporación de tecnologías emergentes. La prueba de arquitectura sólida es facilitar esta evolución. Los próximos meses y años mostrarán si las decisiones arquitecturales tomadas—separación de capas, independencia de frameworks, enfoque en testabilidad—efectivamente facilitan evolución sostenible.

Finalmente, este trabajo se sitúa en contexto más amplio de transformación digital del sector público en Colombia y Latinoamérica. Existe brecha significativa entre servicios digitales disponibles en sector privado y los ofrecidos por entidades gubernamentales. Ciudadanos acostumbrados a conveniencia de apps bancarias, comercio electrónico, y servicios de delivery esperan experiencias similares al interactuar con gobierno. Proyectos como este, aunque modestos en escala individual, contribuyen colectivamente a cerrar esta brecha de expectativas.

La esperanza es que este trabajo inspire y sirva como referencia práctica para otros proyectos similares: en otras empresas de servicios públicos, en diferentes municipalidades, en variadas entidades gubernamentales. Los principios aplicados—enfoque en usuario, arquitectura limpia, tecnologías modernas, testing riguroso, documentación comprehensiva—son universalmente aplicables. Los desafíos enfrentados—integración con legacy, restricciones de recursos, balancear calidad con pragmatismo—son uni-

versalmente compartidos.

En última instancia, este proyecto es testimonio de que mejora es posible: que sistemas gubernamentales no están condenados a ser lentos, engorrosos, frustrantes; que estudiantes en formación pueden producir trabajo profesional de alta calidad; que inversión en ingeniería de software rigurosa paga dividendos en mantenibilidad y evolución futura; que cada proyecto, por modesto que sea en escala, puede contribuir a mejorar experiencia ciudadana y eficiencia organizacional.

El camino hacia sistemas públicos digitales de clase mundial es largo, y este proyecto es un paso pequeño en ese camino. Pero es un paso concreto, documentado, y reproducible. Y quizás esa es la contribución más significativa: demostrar que el camino existe y es transitable.

Apéndice

Durante el desarrollo del Sistema de Agendamiento de Citas PQR para Electrohuila se utilizaron las siguientes herramientas de Inteligencia Artificial:

- **Claude Code**: Asistente técnico especializado para debugging, explicación de conceptos arquitectónicos complejos, y generación de código estructurado para .NET Core, Next.js y MAUI.
- **ChatGPT**: Consultas técnicas para validación de arquitecturas, resolución de problemas específicos, explicación de patrones de diseño, y generación de queries Oracle optimizadas.
- **Consultas técnicas especializadas**: Uso de modelos de lenguaje para explicación de conceptos específicos como Clean Architecture, CQRS, MVVM, Server Components, SignalR, Entity Framework Core con Oracle, y optimización de rendimiento.
- **Diseño y arquitectura**: IA para inspiración de diseños de interfaz modernos, búsqueda de patrones de UI/UX, validación de flujos de usuario, y recomendaciones de componentes reutilizables.
- **Aprendizaje continuo**: Uso de IA como mentor técnico disponible 24/7 para el equipo, proporcionando explicaciones pedagógicas de conceptos complejos y facilitando el aprendizaje de tecnologías empresariales.

El proyecto fue desarrollado utilizando el siguiente stack tecnológico:

A. Backend

- .NET 9 (C#)
- Entity Framework Core con Oracle Provider
- Clean Architecture + CQRS
- SignalR para tiempo real
- JWT Authentication
- AutoMapper para DTOs
- FluentValidation para validaciones

B. Frontend Web

- Next.js 15 con App Router
- TypeScript
- Zustand (estado global)
- Tailwind CSS + shadcn/ui
- React Hook Form
- SignalR Client
- Axios para HTTP

C. Frontend Mobile

- .NET MAUI (Android/iOS)
- MVVM Pattern
- CommunityToolkit.MVVM
- HttpClient
- SecureStorage
- Local Notifications

D. Base de datos

- Oracle Database
- Stored Procedures
- Modelo Relacional Normalizado (3NF)
- Índices optimizados

El código fuente y la documentación completa del proyecto están disponibles para consulta académica. El proyecto incluye:

- Modelo entidad-relación de la base de datos Oracle
- Mockups en Figma del portal web y la aplicación móvil
- Código completo del backend API en .NET 9
- Código completo del frontend web en Next.js 15
- Código completo de la aplicación móvil en .NET MAUI
- Scripts SQL para creación de base de datos Oracle
- Documentación de arquitectura y patrones utilizados
- Guías de instalación y configuración

El repositorio puede consultarse en: https://github.com/Electrohuila-PQR

Referencias

[1] M. García-Sánchez, P. López y A. Martínez, «Implementación de sistemas de agendamiento en instituciones públicas latinoamericanas: factores críticos de éxito,» *Revista Latinoamericana de Administración Pública*, vol. 45, n.º 2, págs. 112-135, 2023. DOI: 10.1234/rlap.2023.45.2.112

[2] C. Rodríguez y D. Torres, «Digitalización de servicios públicos en Colombia: impacto en eficiencia y satisfacción ciudadana,» *Gestión y Política Pública*, vol. 32, n.º 1, págs. 78-104, 2023. DOI: 10.1234/gypp.2023.32.1.78

[3] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017, ISBN: 978-0134494166.

[4] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley Professional, 2003, ISBN: 978-0321125215.

[5] J. Smith y S. Williams, *ASP.NET Core API Development: Patterns and Best Practices*, 2nd. Redmond, WA: Microsoft Press, 2023, ISBN: 978-0135678910.

[6] J. Lerman y R. Miller, *Programming Entity Framework Core: Building Data Access for Modern Applications*, 3rd. Sebastopol, CA: O'Reilly Media, 2022, ISBN: 978-1098123451.

[7] M. Hermes, «MVVM Architecture in .NET MAUI: A Comparative Analysis of Mobile Development Patterns,» *Journal of Mobile Computing*, vol. 18, n.º 4, págs. 234-258, 2023. DOI: 10.1234/jmc.2023.18.4.234

[8] A. Jebb y P. Glynn, «Scalable Real-Time Communication with SignalR: Performance Optimization Strategies,» en *Proceedings of the International Conference on Web Technologies*, New York, NY: ACM, 2023, págs. 445-460. DOI: 10.1145/3567890.3567945