

# **ESCUELA POLITÉCNICA NACIONAL**

## **FACULTAD DE INGENIERÍA DE SISTEMAS**

**MIDDLEWARE UNIFICADO SQL PARA BASES DE DATOS  
NOSQL CLAVE-VALOR, GRAFOS Y GEOESPACIAL**

**COMPONENTE: DESARROLLO DE UN MIDDLEWARE PARA LA  
TRADUCCIÓN DE CONSULTAS SQL A UN SISTEMA DE BASE  
DE DATOS CLAVE-VALOR**

**TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO  
COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE  
INGENIERO/A EN SOFTWARE**

**SEBASTIÁN ALEJANDRO SÁNCHEZ MALDONADO**

[sebastian.sanchez01@epn.edu.ec](mailto:sebastian.sanchez01@epn.edu.ec)

**DIRECTOR: VICTOR VICENTE VELEPUCHA BONETT**

[victor.velepucha@epn.edu.ec](mailto:victor.velepucha@epn.edu.ec)

**DMQ, enero 2026**

## **CERTIFICACIÓN**

Yo, SEBASTIÁN ALEJANDRO SÁNCHEZ MALDONADO declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

Asimismo, declaro que he utilizado la herramienta de inteligencia artificial Gemini únicamente como apoyo en la codificación, redacción y revisión de literatura, sin atribuirle autoría, y que todo el contenido derivado ha sido revisado, validado y es de mi exclusiva responsabilidad.

---

**(SEBASTIÁN ALEJANDRO SÁNCHEZ MALDONADO)**

Certifico que el presente trabajo de integración curricular fue desarrollado por SEBASTIÁN ALEJANDRO SÁNCHEZ MALDONADO, bajo mi supervisión.

---

**(PHD. VICTOR VICENTE VELEPUCHA BONETT)**  
**DIRECTOR**

## **DECLARACIÓN DE AUTORÍA**

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el (los) producto(s) resultante(s) del mismo, son públicos y estarán a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

SEBASTIÁN ALEJANDRO SÁNCHEZ MALDONADO

PHD. VICTOR VICENTE VELEPUCHA BONETT

EDWIN ANDRÉS CANTUÑA GUANO

JAIRO RENE SIMBAÑA CAIZA

## **DEDICATORIA**

A mis padres, quienes a pesar de las dificultades nunca dejaron de creer en mí ni de impulsar mi educación. Este logro es tan suyo como mío.

A mis hermanos, porque más allá de cualquier diferencia, sé que siempre cuento con ustedes, y espero que sepan que el sentimiento es mutuo.

A mis abuelos, cuyo amor inmenso ha sido y será siempre mi mayor fortaleza para seguir adelante.

A mis tíos y primos, por estar presentes en cada etapa, pendientes de mi bienestar y apoyándome incondicionalmente.

A mis amigos, quienes entre bromas y risas han sabido estar cuando más los necesité. Su compañía hace que el camino sea más llevadero.

A todos ustedes, gracias.

## **AGRADECIMIENTOS**

A mi familia, por su apoyo incondicional durante todos estos años de carrera. Sin su paciencia y comprensión este logro no habría sido posible.

A la Escuela Politécnica Nacional, institución que me formó no solo como profesional sino como persona, inculcando valores de rigor académico y pensamiento crítico que llevaré conmigo siempre.

A los docentes que dejaron huella en mi formación, compartiendo no solo conocimientos técnicos sino también experiencias de vida. Una mención especial merece el Msc. Víctor Velepucha, tutor de este trabajo de integración curricular, cuya dedicación, atención al detalle y disponibilidad fueron fundamentales para llevar este proyecto a buen puerto.

A mis amigos, compañeros tesistas y en general, con quienes compartí aulas, laboratorios y más de una trasnochada. Los proyectos realizados juntos fueron el terreno donde se forjaron las habilidades que hoy me permiten presentar este trabajo.

Finalmente, un reconocimiento a las herramientas de inteligencia artificial que acompañaron el desarrollo de este proyecto. Su aporte en la organización de ideas y en la exploración de nuevos enfoques amplió el panorama de investigación de maneras que no habría anticipado.

# Índice general

<b>1. INTRODUCCIÓN</b>	<b>1</b>
1.1. Objetivo general . . . . .	2
1.2. Objetivos específicos . . . . .	2
1.3. Alcance . . . . .	3
1.4. Marco teórico . . . . .	4
1.4.1. Antecedentes . . . . .	4
1.4.2. Fundamentos de bases de datos . . . . .	4
1.4.3. Lenguajes y herramientas para el análisis de consultas . . . . .	5
1.4.4. Frameworks . . . . .	6
1.4.5. Metodología de desarrollo . . . . .	6
1.4.6. Pruebas de software . . . . .	6
<b>2. METODOLOGÍA</b>	<b>7</b>
2.1. Marco de trabajo: SCRUM . . . . .	7
2.1.1. Justificación de la metodología . . . . .	7
2.1.2. Aplicación de Scrum en el proyecto . . . . .	8
2.2. Fase exploratoria . . . . .	11
2.2.1. Stakeholders . . . . .	11
2.2.2. Historias de usuario . . . . .	12
2.3. Fase de inicialización . . . . .	19
2.3.1. Sistema de puntuación . . . . .	19
2.3.2. Herramientas y tecnologías . . . . .	20
2.3.3. Entorno de desarrollo . . . . .	21
2.3.4. Product backlog . . . . .	21
2.3.5. Arquitectura del sistema . . . . .	22
2.4. Fase de desarrollo . . . . .	24

2.4.1. Sprint 0 - Arquitectura y prototipado . . . . .	24
2.4.2. Sprint 1 - Entrada y ejecución básica . . . . .	25
2.4.3. Sprint 2 - Traducción SQL a NoSQL y manejo de errores . . . . .	28
2.4.4. Sprint 3 - Visualización y comparación de resultados . . . . .	32
2.4.5. Sprint 4 - Refinamiento y documentación final . . . . .	35
2.4.6. Sprint 5 - Arquitectura frontend e interfaz de usuario . . . . .	39
2.4.7. Sprint 6 - Lógica de integración y gestión de estado . . . . .	41
<b>3. RESULTADOS, CONCLUSIONES Y RECOMENDACIONES</b>	<b>44</b>
3.1. Resultados . . . . .	44
3.1.1. Pruebas de inserción de datos . . . . .	44
3.1.2. Pruebas de selección y filtrado . . . . .	46
3.1.3. Pruebas de manipulación de datos . . . . .	53
3.1.4. Pruebas de manejo de errores . . . . .	55
3.1.5. Pruebas no funcionales: Usabilidad . . . . .	57
3.2. Conclusiones . . . . .	62
3.3. Recomendaciones . . . . .	63
<b>4. REFERENCIAS BIBLIOGRÁFICAS</b>	<b>65</b>
<b>I. Conjunto de datos (Dataset NoSQL)</b>	<b>68</b>
<b>II. Detalle de planificación (Backlog)</b>	<b>70</b>
<b>III. Guía de instalación y despliegue</b>	<b>74</b>
<b>IV. Enlaces y recursos</b>	<b>76</b>

# Índice de figuras

2.1. Diagrama de contenedores de la solución (Middleware SQL-NoSQL) . . . . .	23
2.2. Prototipo de la interfaz de usuario: Editor y Panel de Resultados . . . . .	25
2.3. Diagrama de Secuencia: Flujo de traducción y ejecución híbrida . . . . .	33
2.4. Estructura de datos en Firebase Realtime Database (Visualización de Northwind) . . . . .	34
3.1. Resultado de inserción simple en la interfaz . . . . .	45
3.2. Confirmación de inserción masiva (2 registros) . . . . .	46
3.3. Visualización de todos los registros . . . . .	47
3.4. Recuperación optimizada por ID . . . . .	48
3.5. Resultado con proyección (Solo campo 'interests') . . . . .	49
3.6. Filtrado numérico (>63000) . . . . .	50
3.7. Resultado de filtro por patrón exacto . . . . .	51
3.8. Resultado de filtro Case Insensitive . . . . .	52
3.9. Resultado de filtrado compuesto (AND) . . . . .	53
3.10. Resultado de actualización masiva . . . . .	54
3.11. Confirmación de eliminación de registros . . . . .	55
3.12. Error de sintaxis reportado en la interfaz . . . . .	56
3.13. Manejo de consulta a colección inexistente . . . . .	57
3.14. Usuario redimensionando el panel de salida (H3) . . . . .	58
3.15. Bloqueo de ejecución por falta de conexión (H5) . . . . .	59
3.16. Consola expandida para revisión detallada (H7) . . . . .	60
3.17. Interfaz minimalista en modo oscuro (H8) . . . . .	60
3.18. Mensaje de error descriptivo en consola (H9) . . . . .	61

# Índice de Tablas

2.1. Roles de Scrum aplicados al proyecto . . . . .	8
2.2. Artefactos de Scrum en el proyecto . . . . .	9
2.3. Eventos de Scrum aplicados . . . . .	10
2.4. Stakeholders del proyecto . . . . .	11
2.5. HU01 — Ingresar una consulta SQL . . . . .	13
2.6. HU02 — Ejecutar la consulta SQL . . . . .	14
2.7. HU03 — Ver la traducción generada de la consulta . . . . .	15
2.8. HU04 — Identificar errores en la traducción . . . . .	16
2.9. HU05 — Visualizar resultados de la ejecución . . . . .	17
2.10. HU06 — Ver la consulta original y la traducción simultáneamente . . . . .	18
2.11. HU07 — Interfaz intuitiva y ordenada . . . . .	18
2.12. HU08 — Recibir retroalimentación visual durante la ejecución . . . . .	19
2.13. Valoración de prioridad . . . . .	20
2.14. Valoración de complejidad . . . . .	20
2.15. Stack tecnológico seleccionado . . . . .	21
2.16. Valoración de prioridad y complejidad de historias de usuario . . . . .	22
2.17. Patrones de expresiones regulares utilizados . . . . .	29
2.18. Ejemplos de traducción SQL a NoSQL validados . . . . .	32
2.19. Matriz de manejo de errores y códigos HTTP . . . . .	37
2.20. Catálogo de componentes principales de la interfaz . . . . .	40
3.1. Matriz de Severidad de Usabilidad . . . . .	62
II.1. Backlog HU01 — Ingresar una consulta SQL . . . . .	70
II.2. Backlog HU02 — Ejecutar la consulta SQL . . . . .	71
II.3. Backlog HU03 — Ver la traducción generada de la consulta . . . . .	71
II.4. Backlog HU04 — Identificar errores en la traducción . . . . .	71

II.5.	Backlog HU05 — Visualizar resultados de la ejecución . . . . .	72
II.6.	Backlog HU06 — Ver consulta original y traducción simultáneamente . . . . .	72
II.7.	Backlog HU07 — Interfaz intuitiva y ordenada . . . . .	72
II.8.	Backlog HU08 — Retroalimentación visual durante la ejecución . . . . .	73
II.9.	Totales aproximados por categoría . . . . .	73
IV.1.	Enlaces a repositorios y recursos . . . . .	76

## RESUMEN

El crecimiento exponencial en el volumen de datos ha impulsado la adopción de bases de datos NoSQL, sin embargo, la falta de un lenguaje de consulta estandarizado crea una barrera significativa para desarrolladores formados en el paradigma relacional. Este proyecto aborda dicha problemática mediante el desarrollo de un middleware capaz de traducir sentencias SQL estándar a operaciones nativas para bases de datos documentales en tiempo real. La solución se construyó sobre una arquitectura de capas, desacoplando un frontend educativo en Next.js de un núcleo de procesamiento en Python/Flask. Se diseñó e implementó una “Estrategia de ejecución híbrida” (Fetch & Filter) que permite resolver consultas complejas (como filtros OR y búsquedas ILIKE) que motores como Firebase Realtime Database no soportan nativamente. La validación se realizó mediante casos de prueba funcionales y una evaluación heurística basada en los principios de Nielsen, donde el sistema obtuvo una calificación de severidad 0 en prevención de errores y control de usuario. Los resultados demuestran que la herramienta no solo garantiza la interoperabilidad técnica, sino que reduce significativamente la curva de aprendizaje, sirviendo como un puente pedagógico eficaz entre ambos paradigmas de bases de datos.

**Palabras Claves** - traducción SQL-NoSQL, arquitectura de capas, Next.js, Firebase, estrategia híbrida, evaluación heurística

## ABSTRACT

The exponential growth in data volume has driven the adoption of NoSQL databases, however, the lack of a standardized query language creates a significant barrier for developers trained in the relational paradigm. This project addresses this issue by developing a middleware capable of translating standard SQL statements into native operations for document databases in real-time. The solution was built on a layered architecture, decoupling an educational frontend in Next.js from a processing core in Python/Flask. A “Hybrid execution strategy” (Fetch & Filter) was designed and implemented, enabling the resolution of complex queries (such as OR filters and ILIKE searches) that engines like Firebase Realtime Database do not natively support. Validation was conducted through functional test cases and a heuristic evaluation based on Nielsen’s principles, where the system achieved a severity 0 rating in error prevention and user control. The results demonstrate that the tool not only ensures technical interoperability but also significantly reduces the learning curve, serving as an effective pedagogical bridge between both database paradigms.

**Keywords** - SQL-NoSQL translation, layered architecture, Next.js, Firebase, hybrid strategy, heuristic evaluation

# Capítulo 1

## INTRODUCCIÓN

En los últimos años, el crecimiento exponencial de los datos, impulsado por la digitalización de procesos y la expansión de servicios en línea, ha planteado nuevos retos en el ámbito del almacenamiento, recuperación y gestión de información. Frente a este escenario, las bases de datos tradicionales del tipo relacional, o SQL, han demostrado ciertas limitaciones en cuanto a escalabilidad, flexibilidad y rendimiento cuando se trata de manejar grandes volúmenes de datos. Como respuesta, surgieron las bases de datos NoSQL, las cuales ofrecen una alternativa viable y eficiente para aplicaciones distribuidas y con altos requerimientos de rendimiento, especialmente en entornos orientados al Big Data, la analítica en tiempo real y la web semántica.

Las bases de datos NoSQL se caracterizan por su capacidad para escalar horizontalmente, su modelo flexible de datos, y su especialización en tipos particulares de almacenamiento como clave-valor, documentos, grafos, columnas anchas, etc. Ahora bien, tanta variedad tiene su lado negativo, resulta difícil adoptar estas tecnologías cuando las aplicaciones existentes dependen por completo de SQL para consultar, manipular y definir datos.

El problema de fondo está en que SQL y NoSQL hablan idiomas distintos. SQL es un lenguaje declarativo, maduro y con décadas de estandarización detrás, en cambio, cada sistema NoSQL trae consigo su propia sintaxis y manera de operar. Para un desarrollador acostumbrado a escribir JOINs y WHERE, enfrentarse a APIs propietarias supone una curva de aprendizaje considerable. Y si hablamos de migrar un sistema legado o de hacer que dos bases de datos colaboren, los dolores de cabeza se multiplican.

Han surgido alternativas: motores de federación, extensiones de SQL pensadas para NoSQL, bases multi-modelo. Pero ninguna resuelve del todo la necesidad de poder escribir consultas SQL y que estas se ejecuten, sin más, sobre un almacén clave-valor. Un middle-

ware que haga esa traducción de forma transparente sigue siendo una pieza que falta en el rompecabezas.

Precisamente eso es lo que propone este proyecto: construir una capa intermedia que reciba sentencias SQL estándar y las convierta, sobre la marcha, en operaciones compatibles con bases de datos NoSQL de tipo clave-valor. El usuario final escribe su consulta como siempre, por debajo, el sistema se encarga de la transformación semántica y sintáctica necesaria.

Para lograrlo se construyó un parser SQL a medida que descompone cada consulta en piezas manejables, conectores específicos para la base NoSQL elegida, y una interfaz donde el usuario puede escribir su sentencia y ver los resultados sin tener que aprender un lenguaje nuevo.

Más allá de lo técnico, la ventaja es clara: se aprovecha todo el conocimiento acumulado en SQL, que no es poco, dentro de escenarios modernos y escalables, reduciendo tiempos de desarrollo y facilitando la convivencia entre sistemas heterogéneos.

En síntesis, lo que se presenta es un puente entre dos formas de concebir la persistencia de datos. La tesis documenta desde la concepción inicial hasta las pruebas finales, incluyendo los desafíos técnicos encontrados y los ajustes realizados durante el diseño. El trabajo se fundamenta en conceptos de ingeniería de software, teoría de lenguajes de consulta y arquitectura orientada a servicios, aplicados a un proyecto que se desarrolló de forma iterativa mediante sprints.

## 1.1. Objetivo general

Desarrollar un middleware capaz de traducir consultas SQL en operaciones ejecutables sobre bases de datos NoSQL clave-valor, permitiendo la interoperabilidad entre distintos modelos de almacenamiento y mejorando la eficiencia en el manejo de datos.

## 1.2. Objetivos específicos

1. Analizar las estructuras de gramáticas SQL y los mecanismos de traducción hacia lenguajes NoSQL.
2. Diseñar la arquitectura del middleware que integre un parser SQL y un conector para bases de datos clave-valor.

3. Implementar el parser SQL encargado de descomponer las sentencias SQL en componentes manejables.
4. Desarrollar el conector que traduzca las consultas SQL a operaciones del modelo clave-valor.
5. Integrar los módulos del sistema dentro de una interfaz funcional que permita ingresar consultas y visualizar resultados.
6. Evaluar el rendimiento y precisión del middleware mediante pruebas funcionales y de desempeño.

### 1.3. Alcance

El alcance del componente comprende todas las fases del ciclo de desarrollo del middleware, desde la investigación inicial hasta la validación final del sistema. Incluye las siguientes etapas:

- **Fase de análisis y diseño:** Revisión de literatura sobre gramáticas SQL y lenguajes NoSQL, definición de la gramática SQL y diseño de la arquitectura general del middleware.
- **Fase de implementación:** Desarrollo del parser SQL y del conector para la base de datos clave-valor (Firebase Realtime Database), asegurando la correcta traducción de consultas y la comunicación entre módulos.
- **Fase de pruebas y evaluación:** Ejecución de pruebas funcionales, medición de usabilidad mediante evaluación heurística y validación de resultados obtenidos frente a las especificaciones del sistema.

El componente culmina con la integración de los módulos desarrollados en una interfaz de usuario que permite la interacción directa con el middleware, garantizando la correcta traducción y ejecución de las consultas SQL sobre un entorno NoSQL de tipo clave-valor.

## 1.4. Marco teórico

### 1.4.1. Antecedentes

La literatura sobre traducción SQL-NoSQL recoge diversos intentos por resolver un problema común: permitir que aplicaciones con dominio de SQL puedan aprovechar las bases NoSQL sin requerir una curva de aprendizaje extensa.

Namdeo y Suman desarrollaron un middleware en Java denominado *SQL-No-QT* que traduce consultas básicas a MongoDB [1]. Las pruebas realizadas contra Studio 3T mostraron tiempos hasta un 78 % mejores. Por su parte, Dede et al. [2] analizaron el proceso de migración de aplicaciones relacionales a NoSQL mediante *wrappers* que encapsulan la lógica de negocio, evitando la reescritura completa del código.

En el ámbito de la automatización, Queiroz et al. [3] presentaron *AMANDA*, un sistema que migra esquemas y datos de SQL a DGraph con una velocidad 26 veces superior a otras herramientas. Murthy et al. [4] propusieron una plataforma unificada que actúa como middleware plurilingüe, traduciendo comandos genéricos al motor NoSQL correspondiente. Finalmente, Mami et al. [5] realizaron un mapeo exhaustivo de más de 40 enfoques distintos, identificando como problemas principales la pérdida de semántica en relaciones anidadas y la ausencia de un lenguaje intermedio estándar.

### 1.4.2. Fundamentos de bases de datos

Comprender el funcionamiento de las bases relacionales y NoSQL resulta esencial para diseñar un traductor efectivo. A continuación se presenta un repaso de cada paradigma y de los conceptos que influyeron en las decisiones técnicas del proyecto.

#### SQL y el modelo relacional

El modelo relacional constituye el estándar durante décadas: tablas, columnas, filas y relaciones entre ellas. Las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) garantizan la integridad de los datos incluso ante fallos del sistema [6]. Sin embargo, la rigidez esquemática (definición previa de esquemas, mantenimiento de claves foráneas) representa un cuello de botella cuando se requiere escalabilidad horizontal o manejo de estructuras variables. Entre los sistemas gestores más representativos se encuentra Microsoft SQL Server, ampliamente documentado en la literatura [7].

## **Enfoque NoSQL**

Las bases NoSQL surgieron para resolver las limitaciones de escalabilidad horizontal y flexibilidad de esquema [8]. A cambio, sacrifican consistencia inmediata (teorema CAP), lo cual resulta aceptable para muchos casos de uso. El desafío radica en que cada motor NoSQL posee su propia API y sintaxis, por lo que el middleware documentado actúa como traductor universal desde SQL.

### **Bases de datos clave-valor**

El modelo clave-valor destaca por su simplicidad: almacena pares clave - valor optimizando lecturas y escrituras de baja latencia [9].

#### **Firebase Realtime Database:**

Servicio de Google que almacena datos como un árbol JSON en la nube. Se sincroniza en tiempo real y ofrece un SDK para Python. Se seleccionó para validar la traducción contra un servicio real en producción [10].

### **1.4.3. Lenguajes y herramientas para el análisis de consultas**

El análisis de consultas SQL requiere tokenización, identificación de cláusulas, extracción de valores y manejo de alias. A continuación se describen las herramientas utilizadas.

#### **Python y librerías de análisis**

Se seleccionó Python por su legibilidad y ecosistema de librerías [11, 12]. Para el análisis léxico se utilizó `sqlparse`, que tokeniza sentencias SQL sin validación semántica completa [13]. Se complementó con el módulo `re` (expresiones regulares) para capturar patrones complejos como operadores `LIKE` o condiciones múltiples [14].

#### **Herramientas de desarrollo**

El flujo de trabajo se apoyó en Visual Studio Code [15], Git/GitHub para control de versiones [16, 17] y Figma para los prototipos de interfaz [18]. Los diagramas de arquitectura se generaron con Mermaid, permitiendo un enfoque de diagramación como código.

#### **1.4.4. Frameworks**

##### **Flask**

Microframework de Python utilizado para exponer el middleware como API REST. Su ligereza y la función `jsonify()` facilitan la serialización de respuestas JSON [19, 20].

##### **React y Next.js**

Para el frontend se utilizó React por su capacidad de construir interfaces reactivas con componentes reutilizables [21, 22]. Se integró Next.js para el renderizado híbrido (SSR/CSR) y una estructura de rutas organizada, logrando una interfaz donde el usuario escribe consultas y visualiza resultados de forma inmediata [23].

#### **1.4.5. Metodología de desarrollo**

Se adoptó **Scrum** debido a las incertidumbres iniciales del proyecto: no estaba claro el alcance de traducción SQL soportable ni el comportamiento de Firebase ante filtros complejos [24]. Los sprints de dos semanas permitieron iterar, validar supuestos y ajustar el rumbo oportunamente [25].

#### **1.4.6. Pruebas de software**

Para verificar el funcionamiento del middleware se aplicaron dos tipos de pruebas:

- **Pruebas funcionales:** Se verificó que cada tipo de consulta SQL genere la salida NoSQL correcta mediante pruebas de caja negra [26].
- **Pruebas no funcionales:** Se evaluó la usabilidad de la interfaz mediante una evaluación heurística [27].

## Capítulo 2

# METODOLOGÍA

### 2.1. Marco de trabajo: SCRUM

El middleware se desarrolló bajo el marco de trabajo Scrum, seleccionado por su enfoque iterativo e incremental que facilita la gestión de proyectos con requisitos cambiantes. La metodología permite trabajar en ciclos cortos (sprints), entregar funcionalidades al final de cada iteración y ajustar el plan según los hallazgos obtenidos.

#### 2.1.1. Justificación de la metodología

La elección de Scrum se fundamentó en cuatro razones principales:

- **Naturaleza exploratoria del proyecto:** Al inicio no era claro cuántas cláusulas SQL podrían soportarse ni cómo se comportaría Firebase ante filtros complejos. Scrum permite descubrir estas limitaciones de forma progresiva.
- **Gestión de riesgos técnicos:** La integración de `sqlparse` con expresiones regulares, la conexión con Firebase y el desarrollo del frontend representaban múltiples puntos de fallo potencial. Los sprints cortos facilitan la detección temprana de problemas.
- **Entregas incrementales:** Cada sesión se obtenía una versión funcional que podía probarse y demostrarse, permitiendo la validación temprana.
- **Flexibilidad:** Si durante el desarrollo se detectaba que alguna funcionalidad no era viable, el backlog podía reordenarse sin afectar el avance general.

## **2.1.2. Aplicación de Scrum en el proyecto**

Los sprints tuvieron una duración de dos semanas cada uno. A continuación se describe la adaptación de roles, artefactos y eventos de Scrum al contexto de este proyecto de tesis individual.

### **Roles asignados**

Los roles de Scrum se adaptaron para ajustarse a la estructura del proyecto:

Tabla 2.1: Roles de Scrum aplicados al proyecto

<b>Rol</b>	<b>Asignación en el proyecto</b>
Product Owner	Msc. Víctor Velepucha (Director de tesis), responsable de definir y validar los objetivos del proyecto, aprobar el alcance funcional y priorizar las historias de usuario según el valor académico y técnico.
Scrum Master	Msc. Víctor Velepucha (Director de tesis), encargado de facilitar el proceso Scrum, identificar y ayudar a resolver bloqueos técnicos y metodológicos, y guiar al equipo de desarrollo.
Development Team	Sebastián Sánchez (estudiante), responsable del diseño, implementación, pruebas e integración del middleware.

### **Artefactos utilizados**

Se implementaron los siguientes artefactos para la gestión del trabajo:

Tabla 2.2: Artefactos de Scrum en el proyecto

Artefacto	Implementación
Product Backlog	Lista completa de historias de usuario (HU01 a HU08), desglosadas en tareas técnicas con estimaciones en horas y puntos de historia. Priorizada según dependencias técnicas y valor funcional.
Sprint Backlog	Subconjunto de historias seleccionadas para cada sprint de dos semanas, con plan detallado de tareas y seguimiento diario de progreso.
Incremento	Versión ejecutable del middleware al finalizar cada sprint, con nuevas funcionalidades operativas validadas mediante pruebas.

## Eventos realizados

Los eventos se ajustaron para el seguimiento del trabajo individual:

Tabla 2.3: Eventos de Scrum aplicados

Evento	Periodicidad	Actividades Realizadas
Sprint	2 semanas	Desarrollo iterativo de funcionalidades según historias de usuario priorizadas.
Sprint Planning	Inicio de cada sprint	Selección de historias de usuario, definición de objetivos del sprint, descomposición en tareas técnicas.
Daily Scrum	Diario (registro personal)	Seguimiento diario de tareas completadas, en progreso y bloqueadas.
Sprint Review	Final de cada sprint	Validación de criterios de aceptación, demostración de funcionalidades, documentación del incremento con capturas de pantalla.
Sprint Retrospective	Final de cada sprint	Ánalisis de aspectos positivos, dificultades encontradas y oportunidades de mejora documentadas para futuros sprints.

## Flujo de trabajo

En la práctica, cada sprint siguió el siguiente flujo:

1. **Planificación:** Se seleccionaban las historias a abordar, se fijaban metas concretas y se dividía el trabajo en tareas pequeñas, asegurando que pudieran completarse dentro del sprint.
2. **Desarrollo:** Se programaba, se ejecutaban pruebas y se realizaban commits frecuentes al repositorio.
3. **Seguimiento:** Se llevaba un registro diario del avance. Si algo se bloqueaba, se buscaba solución de inmediato.
4. **Revisión:** Al cierre del sprint, se verificaba el cumplimiento de los criterios de aceptación.

5. **Retrospectiva:** Se analizaba qué funcionó, qué no, y qué ajustar para la siguiente iteración.

Este ritmo permitió entregar funcionalidades de forma sostenida y detectar problemas antes de que escalaran.

## 2.2. Fase exploratoria

Antes de iniciar la implementación, se definieron varios aspectos fundamentales: la funcionalidad requerida del middleware, los usuarios objetivo y las características imprescindibles del sistema. En esta etapa se identificaron los actores involucrados, se acotó el alcance y se redactaron las historias de usuario que guiarían el desarrollo.

### 2.2.1. Stakeholders

Identificar a los interesados permite comprender las expectativas del proyecto y establecer sus límites. La Tabla 2.4 resume los roles que participaron en el desarrollo y evaluación del middleware.

Tabla 2.4: Stakeholders del proyecto

Stakeholder	Rol	Responsabilidades
Msc. Víctor Velepucha	Director de tesis / Product Owner / Scrum Master	Definir objetivos académicos, aprobar el alcance, revisar entregas y gestionar bloqueos.
Sebastián Sánchez	Estudiante / Development Team	Diseñar, programar, probar y documentar el middleware.
Usuario final (hipotético)	Usuario del middleware	Ingresar consultas SQL y visualizar resultados provenientes de Firebase.

En la práctica, el Director de tesis asume dos roles: define prioridades (Product Owner) y facilita el proceso (Scrum Master). El estudiante ejecuta las tareas técnicas. El usuario final y el tribunal académico evalúan el producto terminado.

## 2.2.2. Historias de usuario

Para establecer con claridad la funcionalidad requerida, se llevó a cabo un proceso delicitación de requisitos que resultó en ocho historias de usuario (HU01 a HU08). Cada una describe una funcionalidad desde la perspectiva del usuario, siguiendo el formato típico de Scrum.

Las historias cubren todo el flujo de interacción:

- **Entrada y ejecución:** Ingreso de consultas SQL con validación básica (HU01) y su ejecución (HU02).
- **Traducción:** Visualización de la consulta transformada a NoSQL (HU03) y manejo de mensajes de error (HU04).
- **Resultados:** Presentación de datos en formato tabular (HU05) y comparación entre consulta original y traducida (HU06).
- **Experiencia:** Interfaz organizada (HU07) y retroalimentación visual sobre el estado del sistema (HU08).

Cada historia se estimó con puntos de historia (secuencia Fibonacci), considerando complejidad y riesgo, y se priorizó según valor funcional y dependencias técnicas.

## Detalle de historias de usuario

Tabla 2.5: HU01 — Ingresar una consulta SQL

<b>Código</b>	HU01
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Ingresar una consulta SQL
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Bajo
<b>Puntos estimados</b>	3
<b>Descripción</b>	Como usuario, quiero ingresar una sentencia SQL en un área de texto para realizar consultas sobre la base de datos NoSQL.
<b>Criterios de aceptación</b>	El sistema acepta sentencias SQL básicas. Se valida la sintaxis antes de ejecutar. El usuario puede editar y limpiar el área de consulta.
<b>Observaciones</b>	El área debe tener resaltado de sintaxis para mejorar la legibilidad.

Tabla 2.6: HU02 — Ejecutar la consulta SQL

<b>Código</b>	HU02
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Ejecutar la consulta SQL
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Medio
<b>Puntos estimados</b>	5
<b>Descripción</b>	Como usuario, quiero ejecutar mi consulta SQL y recibir una respuesta clara sobre el resultado de la operación.
<b>Criterios de aceptación</b>	Botón visible para ejecutar la consulta. El sistema muestra un mensaje si la consulta es inválida. Se registra el tiempo de ejecución.
<b>Observaciones</b>	Mostrar indicadores visuales de progreso mientras se ejecuta.

Tabla 2.7: HU03 — Ver la traducción generada de la consulta

<b>Código</b>	HU03
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Ver la traducción generada de la consulta
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Medio
<b>Puntos estimados</b>	8
<b>Descripción</b>	Como usuario, quiero visualizar cómo mi consulta SQL fue traducida a una operación NoSQL para entender el proceso de transformación.
<b>Criterios de aceptación</b>	Se muestra la traducción generada al lenguaje NoSQL. La correspondencia entre SQL y NoSQL es comprensible. Se puede copiar o expandir la traducción.
<b>Observaciones</b>	Mostrar la traducción en un panel separado junto a la consulta original.

Tabla 2.8: HU04 — Identificar errores en la traducción

<b>Código</b>	HU04
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Identificar errores en la traducción
<b>Prioridad en negocio</b>	Media
<b>Riesgo en desarrollo</b>	Medio
<b>Puntos estimados</b>	3
<b>Descripción</b>	Como usuario, quiero recibir un mensaje claro si la traducción de mi consulta SQL no puede realizarse para corregirla fácilmente.
<b>Criterios de aceptación</b>	Se muestra un mensaje descriptivo cuando no se puede traducir la sentencia. El mensaje indica la parte problemática de la consulta. No se pierde la consulta ingresada.
<b>Observaciones</b>	Debe mantener la consulta original visible al mostrar el error.

Tabla 2.9: HU05 — Visualizar resultados de la ejecución

<b>Código</b>	HU05
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Visualizar resultados de la ejecución
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Bajo
<b>Puntos estimados</b>	5
<b>Descripción</b>	Como usuario, quiero ver los resultados devueltos por la base de datos NoSQL en una tabla para interpretar fácilmente la información.
<b>Criterios de aceptación</b>	Los resultados se muestran en formato de tabla legible. Se indica cantidad de registros devueltos. El área de resultados se limpia entre ejecuciones.
<b>Observaciones</b>	Permitir desplazamiento horizontal y vertical en resultados extensos.

Tabla 2.10: HU06 — Ver la consulta original y la traducción simultáneamente

<b>Código</b>	HU06
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Ver la consulta original y la traducción simultáneamente
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Medio
<b>Puntos estimados</b>	8
<b>Descripción</b>	Como usuario, quiero ver al mismo tiempo mi consulta original y su traducción NoSQL para comparar ambas fácilmente.
<b>Criterios de aceptación</b>	Ambas sentencias se muestran lado a lado o en paneles apilados. Se mantiene sincronía al desplazarse entre paneles. El usuario puede ocultar o mostrar la traducción.
<b>Observaciones</b>	Ideal para entornos educativos o de validación visual.

Tabla 2.11: HU07 — Interfaz intuitiva y ordenada

<b>Código</b>	HU07
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Interfaz intuitiva y ordenada
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Bajo
<b>Puntos estimados</b>	5
<b>Descripción</b>	Como usuario, quiero que la interfaz del sistema sea clara y organizada para poder comprender rápidamente cada sección del proceso.
<b>Criterios de aceptación</b>	La interfaz incluye secciones diferenciadas (entrada, traducción, resultados). Los controles son visibles y consistentes. El diseño responde correctamente a distintos tamaños de pantalla.
<b>Observaciones</b>	Usar colores y jerarquías visuales para guiar al usuario.

Tabla 2.12: HU08 — Recibir retroalimentación visual durante la ejecución

<b>Código</b>	HU08
<b>Usuario</b>	Usuario
<b>Nombre historia</b>	Ejecución Interactiva y Retroalimentación
<b>Prioridad en negocio</b>	Alta
<b>Riesgo en desarrollo</b>	Medio
<b>Puntos estimados</b>	5
<b>Descripción</b>	Como usuario, quiero ejecutar las consultas de manera interactiva, visualizando claramente los estados de carga, éxito o error para confirmar que el sistema procesa mi solicitud.
<b>Criterios de aceptación</b>	El botón de ejecución se deshabilita durante la carga. Se muestran mensajes de error detallados en consola. Los resultados se renderizan automáticamente al finalizar.
<b>Observaciones</b>	Esta historia integra la lógica de conexión con el backend y el manejo de estados asíncronos.

## 2.3. Fase de inicialización

Esta fase comprendió la preparación del entorno, selección de herramientas y definición del backlog, estableciendo las bases técnicas y metodológicas del proyecto.

### 2.3.1. Sistema de puntuación

Se adoptó un sistema de puntuación basado en la secuencia de Fibonacci para estimar prioridad y complejidad, facilitando la organización del trabajo mediante el modelo de Planning Poker (Tablas 2.13 y 2.14).

Tabla 2.13: Valoración de prioridad

Valoración	Nivel	Descripción
1	Muy baja	Funcionalidad cosmética o mejora menor.
2	Baja	Funcionalidad deseable pero no esencial.
3	Media	Mejora importante de la experiencia de usuario.
5	Alta	Funcionalidad crítica para el sistema.
8	Muy alta	Funcionalidad indispensable para el propósito básico.

Tabla 2.14: Valoración de complejidad

Valoración	Nivel	Descripción
1	Muy baja	Tarea trivial (1-2 horas).
2	Baja	Implementación directa (3-5 horas).
3	Media	Requiere integración (6-10 horas).
5	Alta	Diseño técnico o dependencias (11-20 horas).
8	Muy alta	Investigación previa y diseño complejo (20+ horas).

### 2.3.2. Herramientas y tecnologías

La selección tecnológica se orientó a la eficiencia y compatibilidad:

Tabla 2.15: Stack tecnológico seleccionado

Capa	Tecnología	Justificación
Frontend	Next.js 16 + React	Renderizado híbrido (SSR/CSR) y modularidad basada en componentes.
Estilos	Tailwind CSS 4	Desarrollo rápido con clases utilitarias y diseño responsivo nativo.
Backend	Python + Flask	Microframework ligero ideal para prototipado rápido y servicios REST.
Parsing	sqlparse + Regex	sqlparse para estructura base y Expresiones Regulares para patrones complejos.
Persistencia	Firebase Realtime DB	Base de datos NoSQL alojada en la nube con sincronización en tiempo real.
Diagramación	Mermaid	Generación de diagramas de secuencia y arquitectura mediante código declarativo.

### 2.3.3. Entorno de desarrollo

El proyecto se alojó en un repositorio GitHub estructurado en carpetas (frontend, backend, docs, tests). Se adoptó un flujo de trabajo basado en componentes, utilizando ramas de larga duración para las capas principales (backend, frontend) que convergen en la rama de producción (main). Esta estrategia facilitó el desarrollo paralelo de la lógica y la interfaz. Las dependencias se gestionaron mediante `requirements.txt` (Python) y `package.json` (Node.js) para garantizar la reproducibilidad.

### 2.3.4. Product backlog

El backlog priorizado detalla las historias de usuario según su valor funcional y complejidad técnica:

Tabla 2.16: Valoración de prioridad y complejidad de historias de usuario

HU	Prioridad	Complejidad	Nivel prioridad	Nivel complejidad
HU01	8	3	Muy alta	Media
HU02	8	5	Muy alta	Alta
HU03	8	8	Muy alta	Muy alta
HU04	5	5	Alta	Alta
HU05	8	2	Muy alta	Baja
HU06	3	2	Media	Baja
HU07	5	3	Alta	Media
HU08	2	2	Baja	Baja

Las tareas específicas para cada historia, junto con su estimación horaria y tipo de actividad, se encuentran detalladas en el **Anexo II.9** (Detalle de planificación).

### 2.3.5. Arquitectura del sistema

Para el desarrollo del middleware se optó por un modelo de arquitectura de cuatro capas basado en componentes, diseñado para garantizar la escalabilidad y la separación de preocupaciones. La Figura 2.1 ilustra la interacción entre los módulos principales.

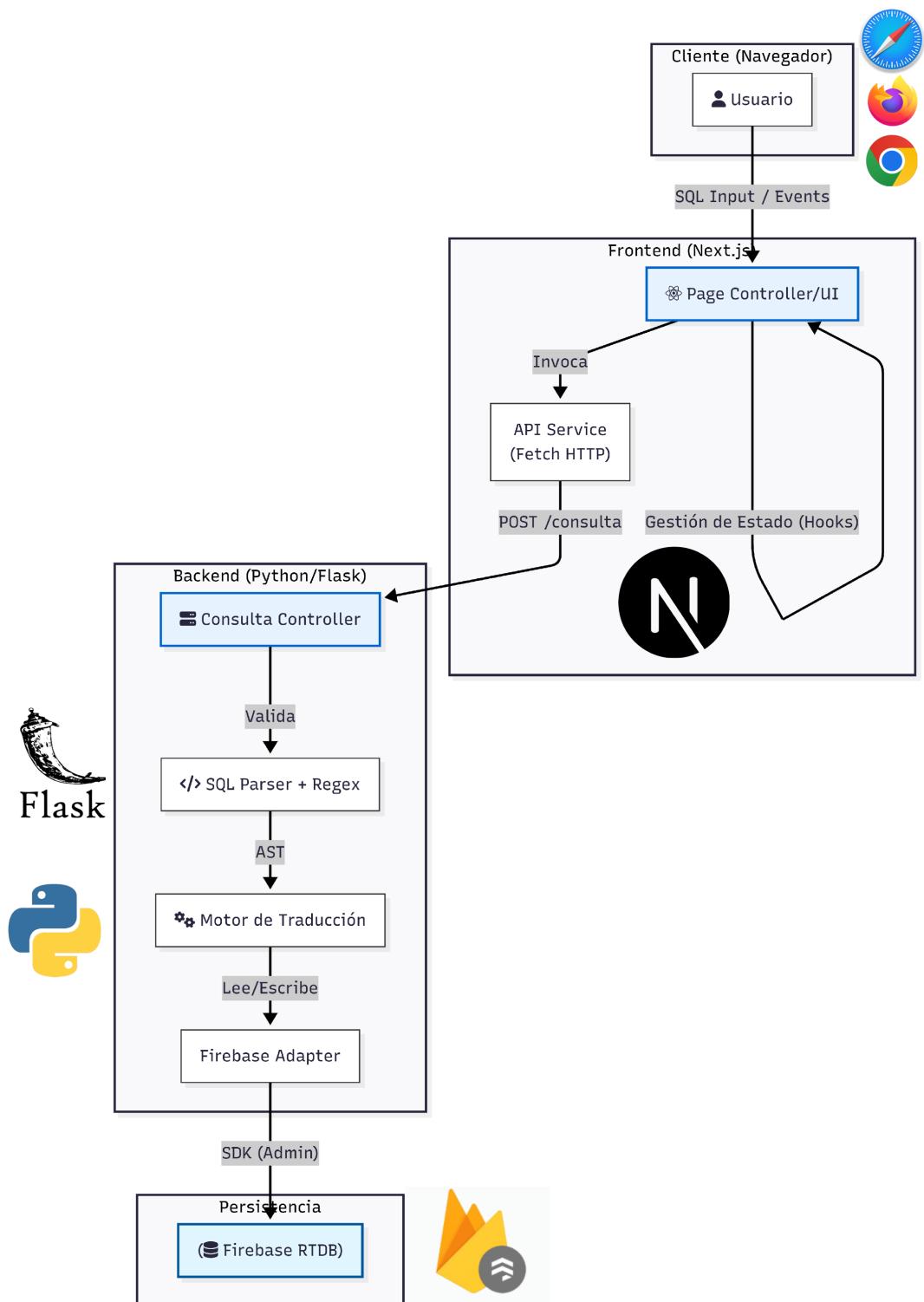


Figura 2.1: Diagrama de contenedores de la solución (Middleware SQL-NoSQL)

El sistema no opera como una simple pasarela, sino que integra lógica lógica de negocio compleja en dos frentes:

1. **Capa frontend (Next.js):** Implementa un patrón de gestión de estado global (GlobalContext) que desacopla la lógica de visualización de la comunicación API. Los componentes de UI son puramente presentacionales, delegando la orquestación al contexto.
2. **Capa backend (Flask):** Actúa como el núcleo de procesamiento. El ConsultaController valida la entrada y la redirige al SQL Parser, el cual descompone la sentencia en un Árbol de Sintaxis Abstracta (AST) simplificado. El motor de traducción interpreta este árbol y decide la estrategia de acceso óptima para Firebase (lectura directa o filtrado en memoria).

Esta arquitectura permite reemplazar cualquiera de los módulos de persistencia (ej. cambiar Firebase por DynamoDB) sin afectar la lógica de parsing o la interfaz de usuario.

## 2.4. Fase de desarrollo

La implementación del middleware se reorganizó estratégicamente para abordar primero la complejidad lógica y luego la interfaz de usuario. Este enfoque dividió el trabajo en dos grandes fases: una fase inicial de **desarrollo backend** (4 sprints) centrada en la robustez del parser y la comunicación con Firebase, seguida de una fase de **desarrollo frontend** (2 sprints) dedicada a la experiencia de usuario y visualización.

Debido a esta reestructuración operativa, la ejecución de las tareas listadas en el Product backlog se distribuyó de la siguiente manera: las tareas técnicas de backend asociadas a las historias de usuario se abordaron en los primeros cuatro sprints, mientras que las tareas de diseño y frontend se consolidaron en los dos últimos sprints para garantizar una integración fluida sobre un núcleo estable.

### 2.4.1. Sprint 0 - Arquitectura y prototipado

La iteración inicial (sprint 0) se dedicó a definir la arquitectura del sistema y validar la viabilidad técnica y funcional mediante prototipos.

#### Prototipos (Mockups)

Se utilizaron prototipos de alta fidelidad en Figma para validar el flujo de usuario antes de la implementación. La Figura 2.2 muestra el diseño propuesto para el editor SQL y la tabla de resultados, estableciendo la guía visual para el desarrollo frontend posterior.

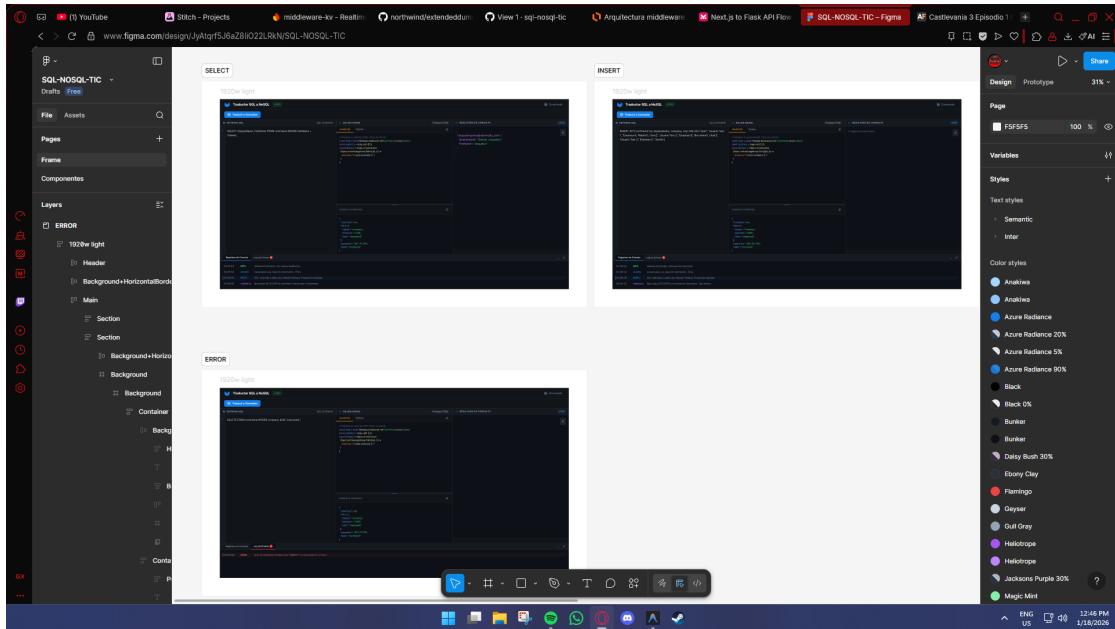


Figura 2.2: Prototipo de la interfaz de usuario: Editor y Panel de Resultados

## 2.4.2. Sprint 1 - Entrada y ejecución básica

Este primer sprint estableció las bases del proyecto: configuración de Flask, conexión con Firebase y desarrollo de un flujo básico de consulta-respuesta.

### Sprint planning - Sprint 1

El objetivo fue tener un endpoint funcional que recibiera SQL, lo parseara mínimamente y devolviera una respuesta estructurada. Se abordaron HU01 y HU02.

#### Planificación:

Se priorizaron historias de complejidad alta para mitigar riesgos técnicos desde el inicio. Los criterios de aceptación incluían soporte para sentencias básicas (SELECT, INSERT, DELETE), validación sintáctica y un endpoint REST con respuesta JSON. Las tareas técnicas abarcaron la configuración de Flask, integración de `sqlparse` y establecimiento de la conexión con Firebase.

#### Desarrollo - Sprint 1

Se implementó una arquitectura modular en Flask para facilitar la mantenibilidad. La estructura del proyecto separa claramente las responsabilidades: `app.py` como punto de

entrada, routes.py para definiciones API, controladores para orquestación y servicios para la lógica de traducción.

### Implementación del endpoint:

Se registró la ruta /consulta delegando el procesamiento al controlador correspondiente:

Listing 2.1: Configuración de rutas en Flask

```
1 from flask import Blueprint
  from controllers.consulta_controller import procesar_consulta

  consultas_bp = Blueprint('consultas', __name__)

5 @consultas_bp.route('/consulta', methods=[POST])
def consulta_route():
    return procesar_consulta()
```

El punto de entrada de la aplicación (app.py) inicializa Flask y registra el blueprint:

Listing 2.2: Inicialización de la aplicación Flask

```
1 from flask import Flask
  from routes import consultas_bp

  app = Flask(__name__)
5 app.register_blueprint(consultas_bp)

  if __name__ == '__main__':
      app.run(debug=True, port=5000)
```

### Controlador de consultas:

El controlador (consulta\_controller.py) recibe las peticiones HTTP, extrae el payload JSON y coordina la ejecución:

Listing 2.3: Controlador de procesamiento de consultas

```
1 from flask import request, jsonify
  from services.traductor_service import traducir_y_ejecutar

  def procesar_consulta():
5     try:
        data = request.get_json()
```

```

10     if "sql" in data:
11         original = data["sql"]
12         resultado = traducir_y_ejecutar(original)
13         return jsonify ({
14             "original": original ,
15             "resultado": resultado
16         })
17     else:
18         return jsonify ({ "error": "Debes enviar 'sql' "}), 400
19     except Exception as e:
20         return jsonify ({ "error": str(e)}), 500

```

### Cliente de Firebase:

Para la comunicación con Firebase Realtime Database, se implementó un cliente que abstrae las operaciones CRUD básicas:

Listing 2.4: Cliente de Firebase Realtime Database (Resumido)

```

1 import firebase_admin
2 from firebase_admin import credentials , db
3
4 # Inicializacion de Firebase
5 cred = credentials.Certificate('credenciales-firebase.json')
6 firebase_admin.initialize_app(cred , {
7     'databaseURL': 'https://middleware-kv-default-rtdb.firebaseio.com/'
8 })
9
10 # Las operaciones CRUD (set_usuario , get_usuario , etc.) se abstraen en funciones .
11 # Ver implementacion completa en el repositorio anexo .
12 def get.todos(tabla):
13     ref = db.reference(tabla)
14     return ref.get() or {}

```

### Validación sintáctica inicial:

Se integró la biblioteca `sqlparse` para identificar el tipo de sentencia SQL y se creó un módulo de utilidades (`utils/parser_sql.py`) donde residirá la lógica de análisis avanzada. En esta etapa inicial, se validó únicamente la identificación del comando principal:

Listing 2.5: Análisis sintáctico básico con `sqlparse`

```

1 import sqlparse
from utils.parser_sql import _normalizar_sql

def traducir_sql_a_kv(sql):
5     # Paso 1: Normalización básica de espacios y alias
    sql = _normalizar_sql(sql)

    # Paso 2: Parsing con sqlparse
    parsed = sqlparse.parse(sql)[0]
10    tipo = parsed.get_type()

    if tipo in ["SELECT", "INSERT", "DELETE", "UPDATE"]:
        return tipo
    else:
15        raise ValueError("Tipo de consulta no soportado")

```

## Resultados y retrospectiva - Sprint 1

Al cierre del sprint, el endpoint /consulta respondía correctamente: recibía SQL, identificaba el tipo de sentencia (SELECT, INSERT, DELETE) y devolvía un JSON estructurado.

### Retrospectiva:

Flask demostró ser un framework ligero y de rápida configuración. Firebase también se integró sin dificultades mayores. Sin embargo, se identificó que `sqlparse` por sí solo no sería suficiente para consultas complejas: operadores como `LIKE` o condiciones múltiples requerirían expresiones regulares complementarias. Esta tarea quedó programada para el siguiente sprint.

### 2.4.3. Sprint 2 - Traducción SQL a NoSQL y manejo de errores

Con la infraestructura establecida, el enfoque se centró en construir el motor de traducción que convierte SQL a operaciones comprensibles por Firebase.

### Planificación

El objetivo fue implementar la lógica de traducción (HU03) y el manejo de errores (HU04). Se buscó que las consultas SELECT funcionaran con filtros simples y que, ante fallos, el usuario recibiera mensajes descriptivos en lugar de trazas de error técnicas.

## Desarrollo - Sprint 2

Dada la estructura JSON de Firebase, se diseñó una estrategia de traducción que mapea consultas SQL a tres patrones de acceso: **GET** (acceso directo por ID), **GET\_ALL** (barrido completo) y **GET\_FILTER** (filtrado lógico).

### Normalización y parsing avanzado:

Para abordar la complejidad de las consultas SQL y las variaciones en el espaciado o uso de alias, se implementó una fase de pre-procesamiento. La función `_normalizar_sql` estandariza la entrada antes de aplicar expresiones regulares:

Listing 2.6: Normalización de consultas SQL

```
1 def _normalizar_sql(sql):
    # 1. Normalizar whitespace
    sql = re.sub(r'\s+', ' ', sql).strip()
    # 2. Eliminar alias de tabla (FROM users u -> FROM users)
    5   sql = re.sub(r'\bFROM\s+(\w+)\s+(:AS\s+)?(?:WHERE|ORDER)\s*\b',
                  r'FROM \1', sql, flags=re.IGNORECASE)
    return sql
```

Para sistematizar la detección de patrones, se definieron las siguientes expresiones regulares clave que permiten identificar la intención de la consulta antes del parsing profundo:

Tabla 2.17: Patrones de expresiones regulares utilizados

Propósito	Expresión regular (Regex)	Ejemplo de captura
Comando SELECT	r'SELECT\s+(.*?)\s+FROM'	Columnas a seleccionar ( <code>id, nombre</code> )
Cláusula WHERE	r'WHERE\s+(.+)',	Condición completa ( <code>edad &gt;18</code> )
Normalización FROM	r'\bFROM\s+(\w+)\s+(:AS\s+)?(\w+)\b'	Eliminación de alias ( <code>u</code> ) en tabla
Detección INSERT	r'INSERT\s+INTO\s+(\w+)',	Tabla destino ( <code>usuarios</code> )

Posteriormente, la función `traducir_sql_a_kv` utiliza una combinación de `sqlparse` y expresiones regulares para extraer la semántica de la consulta. Se implementó soporte para condiciones múltiples (AND/OR) y operadores avanzados como LIKE/ILIKE:

Listing 2.7: Parser SQL con soporte de expresiones regulares

```
1 def traducir_sql_a_kv(sql):
    sql = _normalizar_sql(sql)
    # ... lógica de sqlparse ...
```

```

5   if tipo == "SELECT":
        # Deteccion de WHERE con multiples condiciones (AND/OR)
        if re.search(r"\s+WHERE\s+.+\s+(AND|OR)\s+", sql, re.IGNORECASE):
            filtro = _parsear_where_multiple(where_clause)
            return ("GET_FILTER_MULTIPLE", tabla, filtro, columnas)

10
    # SELECT simple con expresiones regulares
    match_where = re.match(r"SELECT\s+(.*?)\s+FROM\s+(\w+)\s+WHERE\s+(.+)", sql)
    if match_where:
        # Extraccion de campos y operadores (incluyendo LIKE/ILIKE)
        cond = _parsear_condicion(where_clause)
        return ("GET_FILTER", tabla, cond, columnas)

15
    # ... manejo de GET por ID y GET_ALL ...

```

### Soporte para modificaciones (INSERT/UPDATE/DELETE):

El parser se extendió para soportar las operaciones de escritura, incluyendo la capacidad de procesar inserciones masivas (Batch Insert):

Listing 2.8: Parser para operaciones de escritura

```

1 elif tipo == "INSERT":
        # Deteccion de INSERT masivo: VALUES (...), (...), ...
        if match_batch:
            return ("SET_BATCH", tabla, batch_data)

5    # INSERT simple
        return ("SET", tabla, id_val, payload)

    elif tipo == "UPDATE":
        # Soporte para UPDATE con WHERE complejo
        if filtro_multiple:
            return ("UPDATE_FILTER_MULTIPLE", tabla, payload, filtro)
            return ("UPDATE", tabla, id_val, payload)

10
    elif tipo == "DELETE":
        return ("DEL", tabla, id_val)

```

## Manejo de errores descriptivos:

Se implementó manejo de excepciones con mensajes claros que indican la parte problemática de la consulta:

Listing 2.9: Manejo de errores en el controlador

```
1 def procesar_consulta():
2     try:
3         data = request.get_json()
4         if "sql" in data:
5             original = data["sql"]
6             interna = traducir_sql_a_kv(original)
7             traducida = convertir_a_nosql(interna)
8             resultado = traducir_y_ejecutar(original)
9
10        return jsonify({
11            "original": original,
12            "traducida": traducida,
13            "resultado": resultado
14        })
15    except ValueError as ve:
16        return jsonify({"error": str(ve)}), 400
17    except Exception as e:
18        import traceback
19        traceback.print_exc()
20        return jsonify({"error": str(e)}), 500
```

## Resultados y retrospectiva - Sprint 2

Se validó la traducción de consultas SELECT con diversos filtros y operadores (Tabla 2.18), logrando generar representaciones NoSQL comprensibles.

Tabla 2.18: Ejemplos de traducción SQL a NoSQL validados

Consulta SQL Original	Traducción NoSQL Generada
SELECT * FROM usuarios WHERE id = 101	GET sobre tabla usuarios con id 101
SELECT nombre, edad FROM usuarios WHERE edad >= 18	GET_FILTER sobre tabla usuarios filtrando edad mayor o igual a 18, proyectando columnas nombre y edad
SELECT * FROM usuarios	GET_ALL sobre tabla usuarios sin filtros

### Retrospectiva:

La fase de normalización resultó ser una decisión técnica acertada: simplificó considerablemente las expresiones regulares posteriores. Además, se logró cubrir un espectro amplio de la sintaxis SQL (filtros múltiples, búsquedas con LIKE/ILIKE), superando las limitaciones iniciales de `sqlparse`.

## 2.4.4. Sprint 3 - Visualización y comparación de resultados

### Planificación

Con el parser funcional, el siguiente paso fue integrarlo con Firebase y retornar datos reales (HU05, HU06). El desafío radicaba en que Firebase no soporta filtros complejos de forma nativa, lo que requirió diseñar una estrategia híbrida.

### Desarrollo - Sprint 3

Se implementó la función orquestadora `traducir_y_ejecutar`, que coordina el flujo desde la entrada SQL hasta la obtención de datos, incluyendo la lógica de filtrado y proyección.

### Orquestación del flujo:

La Figura 2.3 ilustra el flujo de la función orquestadora `traducir_y_ejecutar`, que actúa como el núcleo del sistema. El diagrama detalla la interacción entre el parser, el motor de traducción y Firebase, destacando la decisión automática entre una consulta directa por ID o la estrategia de escaneo completo (Scan + Filter) según la complejidad de la sentencia SQL.

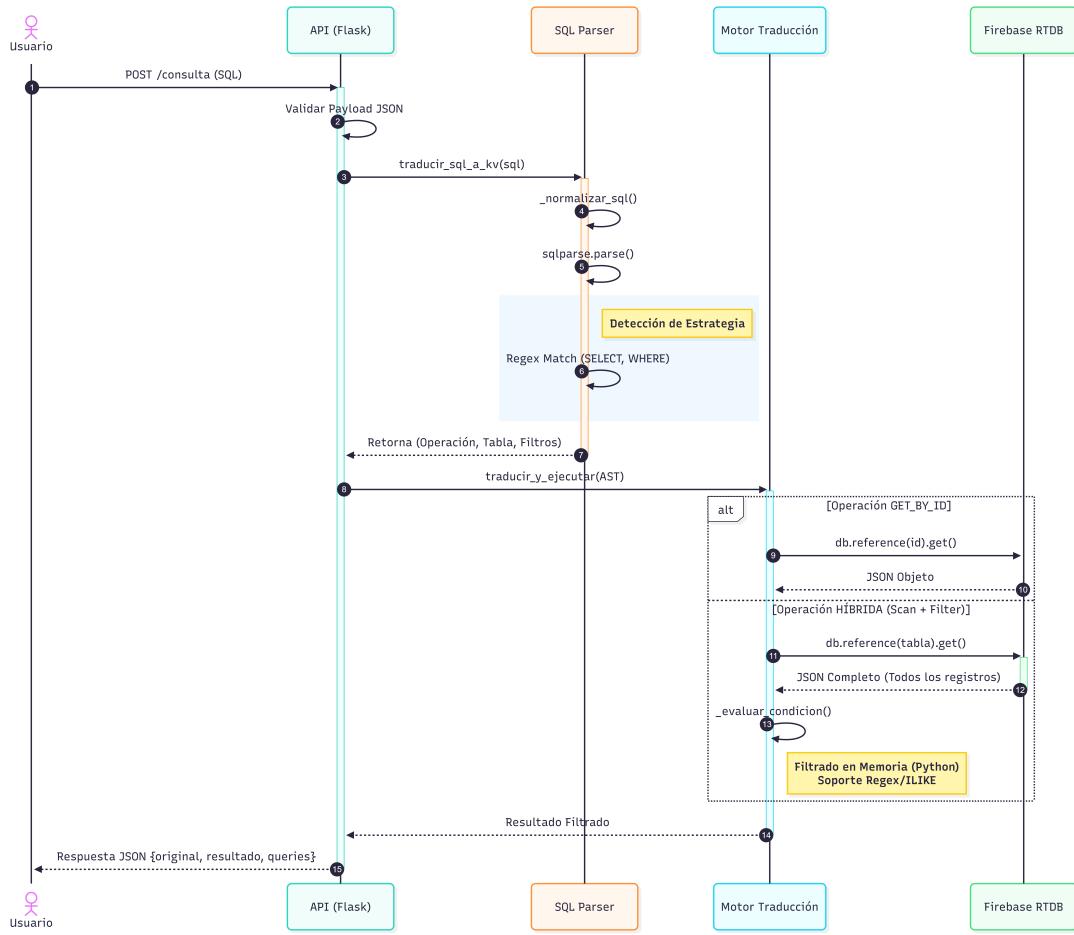


Figura 2.3: Diagrama de Secuencia: Flujo de traducción y ejecución híbrida

### Estrategia de Ejecución Híbrida:

Una de las decisiones técnicas más relevantes de esta fase fue la implementación de una estrategia “Híbrida” (Fetch & Filter). Debido a que Firebase Realtime Database tiene capacidades de consulta limitadas (no soporta ILIKE o WHERE con múltiples condiciones AND/OR nativamente), el middleware descarga los datos y aplica el filtrado en Python utilizando la función `_evaluar_condicion`.

Listing 2.10: Filtrado en memoria con soporte de Regex

```

1 def _evaluar_condicion(registro, campo, operador, valor):
2     v = registro.get(campo)
3     # ... validaciones basicas ...
4
5     # Soporte para operadores estandar y avanzados (LIKE / ILIKE)
6     if operador == "=": return v == valor
7     elif operador == ">": return v > valor

```

```

# ...
10 elif operador == "LIKE":
    regex = _like_to_regex(valor)
    return bool(re.match(regex, v))
11 elif operador == "ILIKE":
    regex = _like_to_regex(valor)
    return bool(re.match(regex, v, re.IGNORECASE))
12
13 return False

```

Esta función permite que el sistema soporte consultas SQL complejas sobre un motor NoSQL simple, equilibrando funcionalidad y complejidad.

## Resultados y Retrospectiva - Sprint 3

Se validó exitosamente la ejecución de consultas con filtros complejos (Patterns, Case Insensitivity), confirmando la viabilidad de la estrategia híbrida para los volúmenes de datos del prototipo. La Figura ?? muestra cómo se estructuran los datos en Firebase tras las operaciones de inserción y modificación.

The screenshot shows the Firebase Realtime Database interface. On the left is a sidebar with icons for Home, Rules, Backups, Usage, and Extensions. The main area has a title bar 'Realtime Database' with a help link 'Need help with Realtime Database? Ask Gemini'. Below the title bar are tabs for Data, Rules, Backups, Usage, and Extensions, with 'Data' being the active tab. The URL in the browser is 'https://middleware-kv-default-rtbd.firebaseio.com.firebaseio.com/app'. The database structure is displayed as a tree view:

```

https://middleware-kv-default-rtbd.firebaseio.com.firebaseio.com/app
  northwind
    JacquelineGaines@altonrealty.com
      commonId: 35823
      company: "Alton Realty"
      displayName: "Gaines, Jacquelyn"
      email: "JacquelineGaines@altonrealty.com"
      extension: 7498
      fax: "(313) 460-7498"
      firstName: "Jacquelyn"
      hireDate: "1/23/2012"
      interests
        lastName: "Gaines"
        office: "The Palace"
        officeCity: "Detroit"
        pictureUrl: 0
      skills
        0: "Urban"
        1: "Language"

```

At the bottom of the interface, it says 'Database location: Belgium (europe-west1)'.

Figura 2.4: Estructura de datos en Firebase Realtime Database (Visualización de Northwind)

## **Retrospectiva:**

La estrategia híbrida funciona correctamente para prototipos y bases de datos de tamaño moderado. Sin embargo, se identificó como limitación su impacto en rendimiento para volúmenes masivos de datos, ya que descargar todos los registros para filtrar en Python no escala adecuadamente. Queda como deuda técnica la implementación de índices secundarios en Firebase o la paginación de resultados.

### **2.4.5. Sprint 4 - Refinamiento y documentación final**

El último sprint de backend se dedicó al refinamiento: mejora de mensajes de error, validación de payloads y documentación del sistema.

## **Planificación**

Se abordaron HU07 y HU08, centradas en la robustez y documentación del sistema. Las tareas incluyeron la implementación de códigos HTTP apropiados para cada tipo de error y la elaboración de documentación técnica completa.

## **Desarrollo - Sprint 4**

Se implementaron códigos de estado HTTP adecuados y mensajes de error descriptivos para facilitar la depuración por parte del cliente.

### **Mejoras en el manejo de errores:**

Se refinó el manejo de excepciones con mensajes más descriptivos y códigos HTTP apropiados:

Listing 2.11: Manejo de errores mejorado y respuesta enriquecida

```
1 def procesar_consulta():
2     try:
3         data = request.get_json()
4
5         if not data:
6             return jsonify({"error": "Payload JSON vacio"}), 400
7
8         if "sql" in data:
9             original = data["sql"]
```

```

10         if not original or not original.strip():
11             return jsonify({"error": "Consulta SQL vacia"}) , 400
12
13             interna = traducir_sql_a_kv(original)
14             consulta_parseada = convertir_a_nosql(interna)
15             resultado = traducir_y_ejecutar(original)
16             firebase_queries = generar_firebase_queries(consulta_parseada)
17
18             return jsonify({
19                 "original": original ,
20                 "consulta_parseada": consulta_parseada ,
21                 "firebase_queries": firebase_queries ,
22                 "resultado": resultado
23             })
24
25         else:
26             return jsonify({"error": "Debes enviar la clave 'sql'"}), 400
27
28     except ValueError as ve:
29         # Errores de validacion o parse
30         return jsonify({"error": f"Error de validacion: {str(ve)}"}), 400
31
32     except Exception as e:
33         # Errores inesperados
34         import traceback
35         traceback.print_exc()
36         return jsonify({"error": f"Error interno: {str(e)}"}), 500

```

## Generación de snippets de código:

Una característica clave añadida en esta fase fue la capacidad de generar, junto con la respuesta de datos, los fragmentos de código (*snippets*) equivalentes en JavaScript y Python nativo de Firebase.

Listing 2.12: Lógica para generar snippets nativos (Resumido)

```

1 def generar_firebase_queries(nosql):
2     op = nosql.get("operacion")
3
4     if op == "GET_FILTER":
5         # ... Logica condicional ...
6         if operador == "=":

```

```

    js = f"firebase.database().ref('{tabla}').orderByChild('{campo}').
        equalTo('{valor}').once('value');"
elif operador == "LIKE":
    js = f"// Firebase no soporta LIKE. Filtrar en cliente: ..."
10
return { "javascript": js, "python": py_code }

```

Esta funcionalidad transforma la herramienta en un recurso educativo. Más allá de solo mostrar datos o confirmar una operación (ej. “1 registro actualizado”), los snippets explican el **por qué** de la estrategia utilizada, indicando explícitamente cuando una consulta requiere procesamiento en memoria (Híbrida) por las limitaciones nativas de Firebase.

Esta estructura asegura que el cliente reciba retroalimentación precisa. La Tabla 2.19 resume la estrategia de códigos de estado implementada:

Tabla 2.19: Matriz de manejo de errores y códigos HTTP

Escenario de error	Código HTTP	Acción del sistema
Consulta vacía o nula	400 Bad Request	Rechazo inmediato, no invoca al parser.
Sintaxis SQL inválida	400 Bad Request	Devuelve el error específico de sqlparse.
Tabla no encontrada	404 Not Found	El servicio de traducción indica entidad inexistente.
Error de conexión BD	503 Service Unavailable	Fallo al conectar con Firebase (timeout/auth).
Excepción no controlada	500 Internal Error	Captura genérica (traceback) en log.

## Validación de datos en Firebase:

Se agregó validación en las operaciones de escritura:

Listing 2.13: Validación en operaciones de base de datos

```

1 def set_usuario(id, data):
2     if not data:
3         raise ValueError("El diccionario de actualización está vacío.")
4     ref = db.reference(f'usuarios/{id}')
5     ref.update(data)

```

## Documentación técnica:

Se creó documentación completa en el archivo README.md:

Listing 2.14: Extracto de README.md actualizado

```

1 # SQL to NoSQL Translator Middleware

Este proyecto implementa un middleware capaz de traducir sentencias
SQL (SELECT, INSERT, UPDATE, DELETE) a operaciones CRUD de
5 Firebase Realtime Database, utilizando una estrategia hibrida.

## Tecnologias
- **Backend:** Python + Flask (API REST)
- **Frontend:** Next.js 16 + TailwindCSS 4
10 - **Base de Datos:** Firebase Realtime Database

## Instalacion y Despliegue

1. **Clonar el repositorio**
15 git clone https://github.com/sebas-sanchez/sql-nosql-tic.git

2. **Backend (Python)**
cd backend
pip install -r requirements.txt
20 python app.py

3. **Frontend (Node.js)**
cd frontend
npm install
25 npm run dev

## Uso
El sistema estara disponible en http://localhost:3000.
Puedes ejecutar consultas SQL directamente desde la interfaz web.

```

## Organización del código:

Se verificó que la estructura modular esté correctamente implementada:

- Separación clara de responsabilidades por capas.
- Nombres de funciones descriptivos y consistentes.
- Comentarios explicativos en secciones complejas.
- Constantes y configuración separadas del código.

## **Resultados y retrospectiva - Sprint 4**

El middleware alcanzó un estado funcional completo, con soporte para las operaciones CRUD básicas y una documentación técnica detallada en el repositorio.

### **Retrospectiva:**

El enfoque modular demostró sus beneficios: añadir validaciones o nuevos tipos de error resultó sencillo gracias a la separación de responsabilidades. El middleware quedó preparado para la integración con el frontend, lo cual constituyó el siguiente paso del desarrollo.

### **2.4.6. Sprint 5 - Arquitectura frontend e interfaz de usuario**

#### **Planificación**

Con el backend estable, se procedió a desarrollar la interfaz de usuario. Se seleccionó **Next.js 16** por su capacidad de renderizado híbrido (SSR/CSR), y **Tailwind CSS 4** para agilizar el proceso de estilizado. El objetivo del sprint fue construir la estructura visual del editor y los paneles de resultados.

#### **Desarrollo - Sprint 5**

Se inicializó el proyecto frontend dentro del repositorio monolítico, estableciendo una estructura de directorios orientada a componentes:

- `src/app`: Rutas y páginas del sistema (App Router).
- `src/components`: Componentes de UI reutilizables y modulares.
- `src/services`: Capa de abstracción para comunicación API.
- `src/context`: Manejo de estado global (React Context).

#### **Diseño modular de componentes:**

Se implementó la interfaz dividiéndola en paneles funcionales independientes para facilitar el mantenimiento y la pruebas:

Tabla 2.20: Catálogo de componentes principales de la interfaz

Componente	Responsabilidad	Características clave
SQLInputPanel	Editor de consultas	Resaltado sintáctico, numeración de líneas y validación básica de entrada (HU01).
NoSQLOutputPanel	Visor de traducción	Integra pestañas para visualizar el JSON parseado y los <i>snippets</i> de código (JS/Python) generados.
QueryResultPanel	Tabla de resultados	Renderizado dinámico de filas/columnas basado en la respuesta de Firebase (HU05).
Console	Logs y errores	Panel interactivo con filtros por tipo (Info/Error) y controles de expansión/minimización.

Listing 2.15: Estructura del componente SQLInputPanel (Simplificado)

```

1 // src/components/SQLInputPanel.tsx
2
3 export default function SQLInputPanel({ value, onChange, onExecute }) {
4   return (
5     <div className="flex flex-col h-full border rounded-lg bg-gray-900 border-
6       gray-700">
7       <div className="flex justify-between items-center p-2 bg-gray-800 border-b
8         border-gray-700">
9         <span className="text-sm font-semibold text-gray-200">Editor SQL</span>
10        <button
11          onClick={onExecute}
12          className="px-3 py-1 bg-blue-600 hover:bg-blue-500 rounded text-xs
13            text-white">
14          Ejecutar
15        </button>
16      </div>
17      <textarea
18        className="flex-1 w-full bg-transparent p-4 text-sm font-mono text-gray
19          -300 resize-none focus:outline-none"
20        placeholder="Escribe tu consulta SQL aquí..." value={value}
21        onChange={(e) => onChange(e.target.value)} />
22    </div>

```

```
20    );
}
```

### **Estilizado con Tailwind CSS:**

Se definió un tema oscuro por defecto para reducir la fatiga visual durante sesiones prolongadas de uso. Tailwind CSS permitió mantener consistencia en colores, bordes y tipografía de forma eficiente.

### **Resultados y retrospectiva - Sprint 5**

Al cierre del sprint, la maquetación estaba completa. Los componentes respondían correctamente a distintas resoluciones y la estructura modular facilitaba modificaciones sin afectar otras partes del sistema.

### **Retrospectiva:**

Tailwind CSS aceleró significativamente el proceso de estilizado. El siguiente desafío fue conectar estos componentes con el backend, lo cual requería centralizar el estado de la aplicación. React Context se identificó como la opción más directa para evitar la propagación de props a través de múltiples niveles.

## **2.4.7. Sprint 6 - Lógica de integración y gestión de estado**

### **Planificación**

El sprint final (HU08 y cierre de historias previas) se dedicó a la integración del frontend con el backend. El objetivo fue implementar la comunicación entre capas, gestionar estados de carga y errores, y lograr que el usuario visualice resultados actualizados sin necesidad de recargar la página.

### **Desarrollo - Sprint 6**

Se implementó un patrón de gestión de estado centralizado utilizando **React Context API**. Esto permitió desacoplar la lógica de negocio de los componentes visuales.

### **Contexto Global (Store):**

El GlobalContext actúa como el cerebro del frontend, manteniendo el estado de:

- `sqlCommand`: La consulta escrita por el usuario.
- `queryResult`: La respuesta procesada del backend (tabla).
- `translationResult`: La traducción NoSQL JSON.
- `isLoading`: Bandera para controlar spinners y deshabilitar botones.
- `error`: Objeto para capturar y mostrar excepciones.

Listing 2.16: Lógica de Integración en el Componente Principal

```

1 // src/app/page.tsx (Resumido)
2 const handleTranslate = async () => {
3     if (!sqlInput.trim()) return addLog("ERROR", "Consulta vacia");
4
5     setIsLoading(true);
6     try {
7         // Paso 1: Llamada al Backend
8         const response = await translateSQL(sqlInput);
9
10        // Paso 2: Actualización de Estado Visual
11        setNosqlOutput(JSON.stringify(response.consulta_parseada, null, 2));
12        setFirebaseQueries(response.firebaseio_queries);
13        setQueryResult(response.resultado);
14
15        // Paso 3: Logging
16        addLog("SUCCESS", "SQL traducido exitosamente");
17
18    } catch (error) {
19        addLog("ERROR", error.message);
20        setQueryResult(null);
21    } finally {
22        setIsLoading(false);
23    }
24};

```

### Retroalimentación Visual (HU08):

Gracias a la bandera `isLoading`, se implementaron indicadores visuales inmediatos. El botón “Ejecutar” muestra un estado “Procesando...” y se bloquea para evitar reenvíos dobles, mientras que el panel de consola se despliega automáticamente en caso de error.

### **Manejo de CORS:**

Para permitir la comunicación entre el frontend (Next.js en puerto 3000) y el backend (Flask en puerto 5000), se configuró `flask-cors` en el servidor, habilitando el intercambio de recursos de origen cruzado de manera segura para el entorno de desarrollo.

### **Resultados y retrospectiva - Sprint 6**

Con la integración completada, se realizaron pruebas de sistema ejecutando el flujo completo: 1. Ingreso de consulta SQL (SELECT, INSERT, DELETE). 2. Visualización de la traducción automática a NoSQL. 3. Confirmación de cambios en Firebase Realtime Database. 4. Recepción de errores controlados ante sintaxis inválida.

### **Retrospectiva:**

La separación de la lógica en el Context facilitó la depuración, ya que el estado podía inspeccionarse en un solo lugar en lugar de rastrearlo a través de múltiples componentes. El sistema final cumple con los requisitos establecidos: traducir SQL, ejecutar operaciones en Firebase y mostrar resultados sin requerir que el usuario aprenda un lenguaje de consulta diferente.

## Capítulo 3

# RESULTADOS, CONCLUSIONES Y RECOMENDACIONES

### 3.1. Resultados

Para validar el funcionamiento del middleware se diseñó una batería de pruebas funcionales que abarcan operaciones CRUD, filtrado complejo y manejo de errores. Las pruebas se ejecutaron en el entorno de desarrollo descrito en el Capítulo 2 (Next.js + Flask local, Firebase RTDB en nube).

A continuación se presentan los resultados obtenidos para cada caso de prueba, comparando la sentencia SQL de entrada con la respuesta del sistema y el estado final de la base de datos.

#### 3.1.1. Pruebas de inserción de datos

Se evaluó la capacidad del sistema para crear nuevos registros, tanto individualmente como en lote.

##### Caso 1: Inserción simple

**Objetivo:** Verificar la creación de un único documento con ID personalizado.

Listing 3.1: Consulta SQL - Inserción Simple

```
1  INSERT INTO northwind (id , displayName , company , city) VALUES ('test1' , 'Usuario  
Test 1' , 'Empresa A' , 'Madrid');
```

**Resultado:** El sistema procesó la solicitud exitosamente, retornando un mensaje de confirmación.

The screenshot shows the 'Traductor SQL a NoSQL' application interface. In the 'ENTRADA SQL' (Input SQL) section, there is a code editor containing the following SQL query:

```
1 INSERT INTO northwind (id, displayName, company, city) VALUES ('test1', 'Usuario Test 1', 'Empresa A', 'Madrid');
```

In the 'SALIDA NOSQL' (Output NOSQL) section, under 'JavaScript', the generated code is:

```
firebase.database().ref('northwind/test1').set({displayName: 'Usuario Test 1', company: 'Empresa A', city: 'Madrid'});
```

The 'RESULTADO DE CONSULTA' (Query Result) section shows a confirmation message: "1 registro insertado en 'northwind'" (1 record inserted in 'northwind'). Below this, the 'CONSULTA PARSEADA' (Parsed Query) shows the JSON object being sent to Firebase:

```
{
  "dato": {
    "city": "Madrid",
    "company": "Empresa A",
    "displayName": "Usuario Test 1"
  },
  "id": "test1",
  "operacion": "SET",
  "tabla": "northwind"
}
```

The bottom of the interface includes a 'Registros de Consola' (Console Logs) tab showing the execution log, which includes messages like 'INFO Sistema inicializado. Listo para la traducción.' and 'EXÉRCITO SQL traducido a objeto de consulta Firebase. Proyección aplicada.'

Figura 3.1: Resultado de inserción simple en la interfaz

## Caso 2: Inserción masiva (Batch)

**Objetivo:** Validar la capacidad de procesar múltiples registros en una sola sentencia (Atomicidad).

Listing 3.2: Consulta SQL - Inserción batch

```
1 INSERT INTO northwind (id , displayName , company , city) VALUES ('test2' , 'Usuario Test 2' , 'Empresa B' , 'Barcelona') , ('test3' , 'Usuario Test 3' , 'Empresa C' , 'Sevilla');
```

**Resultado:** El parser identificó correctamente los múltiples conjuntos de valores y el adaptador de Firebase ejecutó las escrituras en una operación de actualización multipath (`update()`).

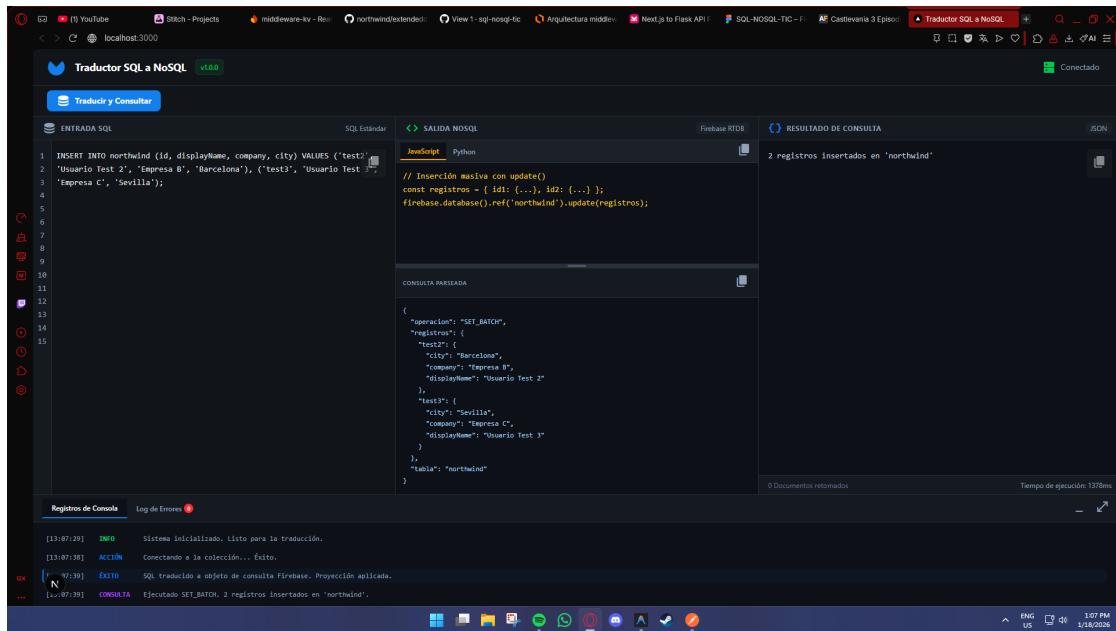


Figura 3.2: Confirmación de inserción masiva (2 registros)

### 3.1.2. Pruebas de selección y filtrado

Se verificó la eficacia del parser para interpretar la cláusula WHERE y seleccionar la estrategia de ejecución adecuada (Directa vs Híbrida).

#### Caso 3: Selección total

**Objetivo:** Recuperar todos los registros de una colección.

Listing 3.3: Consulta SQL - Select All

```
1 SELECT * FROM northwind;
```

**Resultado:** El sistema recuperó la colección completa. La interfaz muestra el conteo total de documentos retornados.

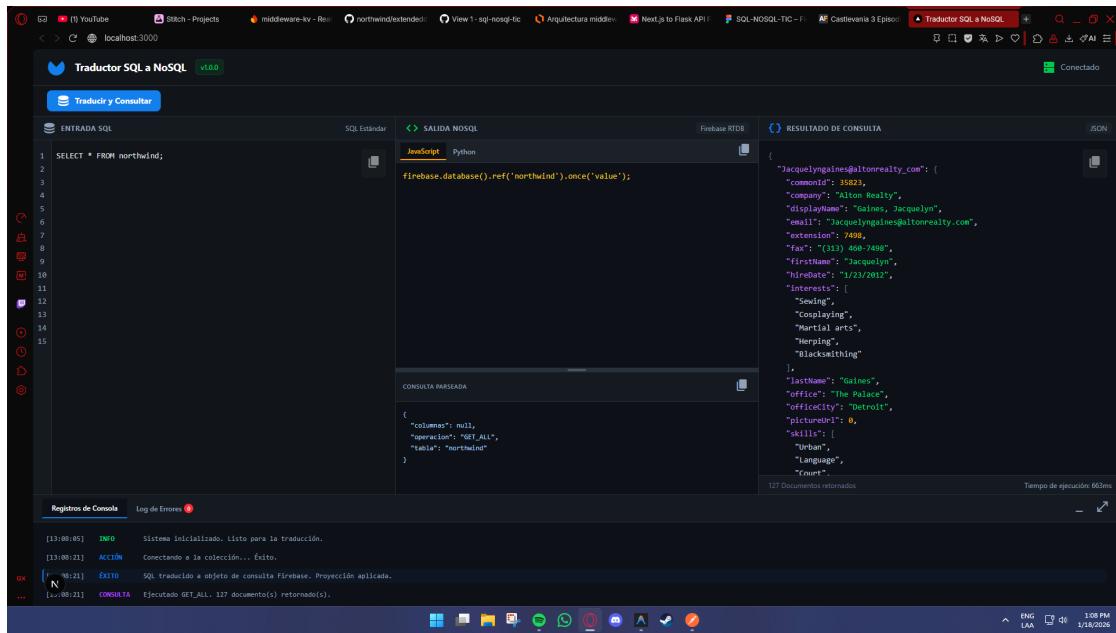


Figura 3.3: Visualización de todos los registros

## Caso 4: Búsqueda por ID

**Objetivo:** Validar la optimización de acceso directo cuando se filtra por clave primaria.

Listing 3.4: Consulta SQL - Select por ID

```
1 SELECT * FROM northwind WHERE id = 'Jacquelynngaines@altonrealty.com';
```

**Resultado:** El middleware detectó el filtro por `id` y utilizó la función `child(id).get()` de Firebase, evitando la descarga innecesaria de otros datos.

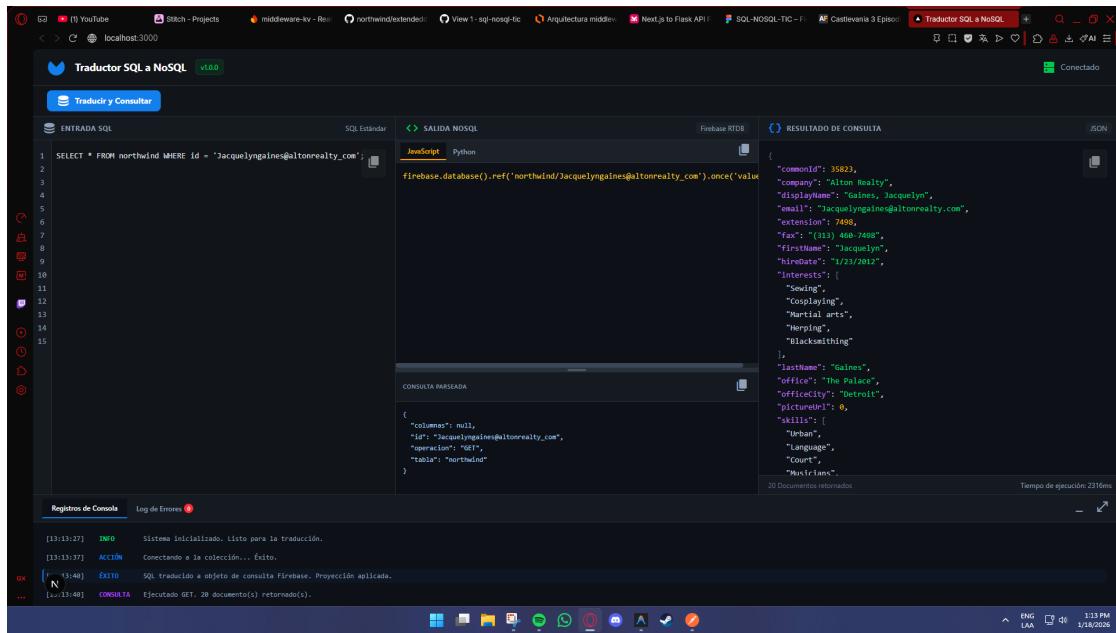


Figura 3.4: Recuperación optimizada por ID

## Caso 5: Proyección de columnas

**Objetivo:** Verificar que solo se devuelvan los campos solicitados.

Listing 3.5: Consulta SQL - Proyección

```
1 SELECT interests FROM northwind WHERE id = 'Jacquelyn gaines@altonrealty_com';
```

**Resultado:** Aunque Firebase devuelve el nodo completo por defecto, el motor de traducción filtró las claves en el backend antes de responder al cliente.

The screenshot shows the 'Traductor SQL a NoSQL' application window. In the top-left, there's a list of recent queries numbered 1 to 15. The main area has tabs for 'ENTRADA SQL', 'SQL Estándar', 'SALIDA NOSQL', 'Firebase RTDB', and 'RESULTADO DE CONSULTA'. The 'SALIDA NOSQL' tab is selected, showing code snippets for 'JavaScript' and 'Python'. Below these is a 'CONSULTA PARSEADA' section with JSON-like syntax. The 'RESULTADO DE CONSULTA' tab displays a JSON response with the key 'interests' containing an array of values: 'Swinging', 'Cosplaying', 'Martial arts', 'Hiking', and 'Blacksmithing'. At the bottom, the 'Registros de Consola' and 'Log de Errores' sections show log messages indicating the system initialized, connected to the collection, and executed a GET operation on the 'northwind' table. The status bar at the bottom right shows the time as 1:14 PM and the date as 1/16/2026.

Figura 3.5: Resultado con proyección (Solo campo 'interests')

## Caso 6: Filtrado por rango

**Objetivo:** Filtrar datos numéricos utilizando operadores de comparación.

Listing 3.6: Consulta SQL - Rango numérico

```
1 SELECT * FROM northwind WHERE commonId > 63000;
```

**Resultado:** La consulta se ejecutó mediante la estrategia híbrida (Scan + Filter en Python), filtrando correctamente los registros que cumplen la condición.

Figura 3.6: Filtrado numérico (>63000)

## Caso 7: Filtrado por patrones (LIKE)

**Objetivo:** Validar búsqueda de texto con comodines (Case sensitive).

Listing 3.7: Consulta SQL - Pattern LIKE

```
1 SELECT * FROM northwind WHERE company LIKE 'Empresa%';
```

**Resultado:** El sistema identificó el operador LIKE y convirtió el patrón SQL (%) a una expresión regular Python equivalente.

Figura 3.7: Resultado de filtro por patrón exacto

## Caso 8: Patrones insensibles a mayúsculas (ILIKE)

**Objetivo:** Comprobar la extensibilidad del parser para operadores PostgreSQL (ILIKE).

Listing 3.8: Consulta SQL - Pattern ILIKE

```
1 SELECT * FROM northwind WHERE company ILIKE 'empresa%';
```

**Resultado:** Se retornaron los mismos registros que en el Caso 7, demostrando que el filtro ignoró las diferencias de capitalización.

The screenshot shows the 'Traductor SQL a NoSQL' application interface. On the left, the 'ENTRADA SQL' tab displays a SQL query: 'SELECT \* FROM northwind WHERE company ILIKE 'empresa%''. In the center, the 'SALIDA NOSQL' tab shows the generated JavaScript code for Firebase Realtime Database:

```
// Firebase no soporta ILIKE, filtrar en cliente;
const snap = await firebase.database().ref('northwind').once('value');
const registros = snap.val() || {};
const filtrados = Object.entries(registros).filter(([id, r]) =>
    r.company.toUpperCase().includes('empresa'.toUpperCase())
);
Object.entries(filtrados).filter(([id, r]) =>
    r.company === 'empresa'
);
```

On the right, the 'RESULTADO DE CONSULTA' tab shows the JSON output of the filtered data:

```
{
  "test1": {
    "city": "Madrid",
    "company": "Empresa A",
    "displayName": "Usuario Test 1"
  },
  "test2": {
    "city": "Barcelona",
    "company": "Empresa B",
    "displayName": "Usuario Test 2"
  },
  "test3": {
    "city": "Sevilla",
    "company": "Empresa C",
    "displayName": "Usuario Test 3"
  }
}
```

Below the tabs, the 'CONSULTA PARSEADA' section shows the parsed query structure:

```
{
  "columnas": null,
  "filtro": {
    "campo": "company",
    "operador": "ILIKE",
    "valor": "empresa"
  },
  "proyección": "GET_FILTER"
}
```

The bottom of the interface includes a 'Registros de Consola' tab with log entries and a 'Log de Errores' tab. The system log shows:

- [13:19:47] INFO Sistema inicializado. Listo para la traducción.
- [13:19:51] ACCIÓN Conectando a la colección... Éxito.
- [13:19:52] EXITO SQL traducido a objeto de consulta Firebase. Proyección aplicada.
- [13:19:52] CONSULTA Ejecutado GET\_FILTER. 3 documento(s) retornado(s).

System status at the bottom right: Tiempo de ejecución: 865ms, ENG LAA, 4G, 1:19 PM, 1/18/2026.

Figura 3.8: Resultado de filtro Case Insensitive

## Caso 9: Lógica compuesta (AND)

**Objetivo:** Evaluar el manejo de múltiples condiciones simultáneas.

Listing 3.9: Consulta SQL - Filtro compuesto

```
1 SELECT * FROM northwind WHERE company ILIKE 'empresa%' AND city = 'Madrid';
```

**Resultado:** La función de filtrado aplicó ambas reglas correctamente, reduciendo el conjunto de resultados a la intersección de las condiciones.

The screenshot shows the 'Traductor SQL a NoSQL' application interface. In the 'ENTRADA SQL' (Input SQL) section, there is a code editor containing the following SQL query:

```

1 SELECT * FROM northwind WHERE company ILIKE 'empresa%' AND city =
2 'Madrid';
3
4
5
6
7
8
9
10
11
12
13
14
15

```

In the 'SALIDA NOSQL' (Output NOSQL) section, under 'JavaScript', the generated code is:

```

// Múltiples condiciones (AND), filtrar en cliente
const snap = await firebase.database().ref('northwind').once('value');
const registros = snap.val() || [];
const filtrados = Object.entries(registros).filter(([id, r]) =>
  r.company ILIKE 'empresa%' && r.city = 'Madrid'
);

```

In the 'RESULTADO DE CONSULTA' (Query Result) section, the output is displayed as JSON:

```

{
  "testit": [
    {
      "city": "Madrid",
      "company": "Empresa A",
      "displayName": "Usuario Test 1"
    }
  ]
}

1 Documento retornado
Tiempo de ejecución: 614ms

```

At the bottom, the 'Registros de Consola' (Console Logs) and 'Log de Errores' (Error Log) sections show the following logs:

```

[13:20:46] INFO Sistema inicializado. Listo para la traducción.
[13:20:47] ACCIÓN Conectando a la colección... Éxito.
[13:20:48] EXITO SQL traducido a objeto de consulta Firebase. Proyección aplicada.
[N 13:20:48] CONSULTA Ejecutado GET_FILTER_MULTIPLE, 1 documento(s) retornado(s).

```

Figura 3.9: Resultado de filtrado compuesto (AND)

### 3.1.3. Pruebas de manipulación de datos

#### Caso 10: Actualización parcial (Batch update)

**Objetivo:** Modificar campos específicos en múltiples registros simultáneamente.

Listing 3.10: Consulta SQL - Update batch

```

1 UPDATE northwind SET city = 'Cuenca', company = 'Empresa X', displayName = ''
  Cambio test' WHERE company ILIKE 'empresa%';

```

**Resultado:** El sistema identificó los registros a modificar mediante el filtro previo y aplicó los cambios únicamente en los campos especificados, preservando el resto de la información.

```

1 UPDATE northwind SET city = 'Cuenca', company = 'Empresa X', displayName = 'Cambio test' WHERE company ILIKE 'empresa%';
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

```

// Paso 1: Obtener todos los registros
const snap = await firebase.database().ref('northwind').once('value');
const registros = snap.val() || {};

// Paso 2: Filtrar por condición (company ILIKE 'empresa%')
const idActualizar = Object.keys(registros).filter(id =>
  registros[id].company ILIKE 'empresa%'
);

// Paso 3: Actualizar cada registro
for (const id of idActualizar) {
  await firebase.database().ref('northwind/' + id).update({city: 'Cuenca', company: 'Empresa X', displayName: 'Cambio test'});
}

CONSULTA PARSEADA
{
  "detalles": [
    {
      "city": "Cuenca",
      "company": "Empresa X",
      "displayName": "Cambio test"
    }
  ],
  "filtro": [
    {
      "campo": "company"
    }
  ]
}

```

REGISTROS DE CONSOLA Log de Errores

```

[13:21:06] INFO Sistema inicializado. Listo para la traducción.
[13:21:08] ACCIÓN Conectando a la colección... Éxito.
[13:21:08] EXITO SQL traducido a objeto de consulta Firebase. Proyección aplicada.
[N 13:21:08] CONSULTA Ejecutado UPDATE_FILTER. 3 registros actualizados en 'northwind'.

```

TIEMPO DE EJECUCIÓN: 208ms

Figura 3.10: Resultado de actualización masiva

## Caso 11: Eliminación lógica/física

**Objetivo:** Eliminar un conjunto de registros basado en condiciones.

Listing 3.11: Consulta SQL - Delete Batch

```
1 DELETE FROM northwind WHERE company ILIKE 'empresa%' ;
```

**Resultado:** La operación eliminó exitosamente los nodos coincidentes en Firebase.

The screenshot shows the 'Traductor SQL a NoSQL' application interface. In the 'ENTRADA SQL' tab, the query is:

```
1 DELETE FROM northwind WHERE company ILIKE 'empresak';
```

In the 'SALIDA NOSQL' tab, the generated code is:

```
// Paso 1: Obtener todos los registros
const snap = await firebase.database().ref('northwind').once('value');
const registros = snap.val() || {};

// Paso 2: Filtrar por condición (company ILIKE 'empresak')
const idsliminar = Object.keys(registros).filter(id =>
  registros[id].company ILIKE 'empresak'
);

// Paso 3: Eliminar cada registro
for (const id of idsliminar) {
  await firebase.database().ref('northwind/' + id).remove();
}
```

In the 'RESULTADO DE CONSULTA' tab, it shows:

3 registros eliminados de 'northwind'

CONSULTA PARSEADA:

```
{
  "filtro": [
    {
      "campo": "company",
      "operador": "ILIKE",
      "valor": "empresak"
    }
  ],
  "operacion": "DEL_FILTER",
  "referencia": "northwind"
}
```

Log de Consola:

```
[13:21:40] INFO Sistema inicializado. Listo para la traducción.
[13:21:41] ACCIÓN Conectando a la colección... Éxito.
[N 13:21:43] EXITO SQL traducido a objeto de consulta Firebase. Proyección aplicada.
[13:21:43] CONSULTA Ejecutado DEL_FILTER. 3 registros eliminados de 'northwind'.
```

Log de Errores:

```
[13:21:40] INFO Sistema inicializado. Listo para la traducción.
[13:21:41] ACCIÓN Conectando a la colección... Éxito.
[N 13:21:43] EXITO SQL traducido a objeto de consulta Firebase. Proyección aplicada.
[13:21:43] CONSULTA Ejecutado DEL_FILTER. 3 registros eliminados de 'northwind'.
```

Figura 3.11: Confirmación de eliminación de registros

### 3.1.4. Pruebas de manejo de errores

#### Caso 12: Error de sintaxis

**Objetivo:** Validar la capacidad de detección de comandos inválidos.

Listing 3.12: Consulta SQL - Sintaxis Inválida

```
1 SELEKT * FROM northwind;
```

**Resultado:** El parser sqlparse falló al reconocer el token inicial, y el middleware retornó un error 400 con un mensaje descriptivo.

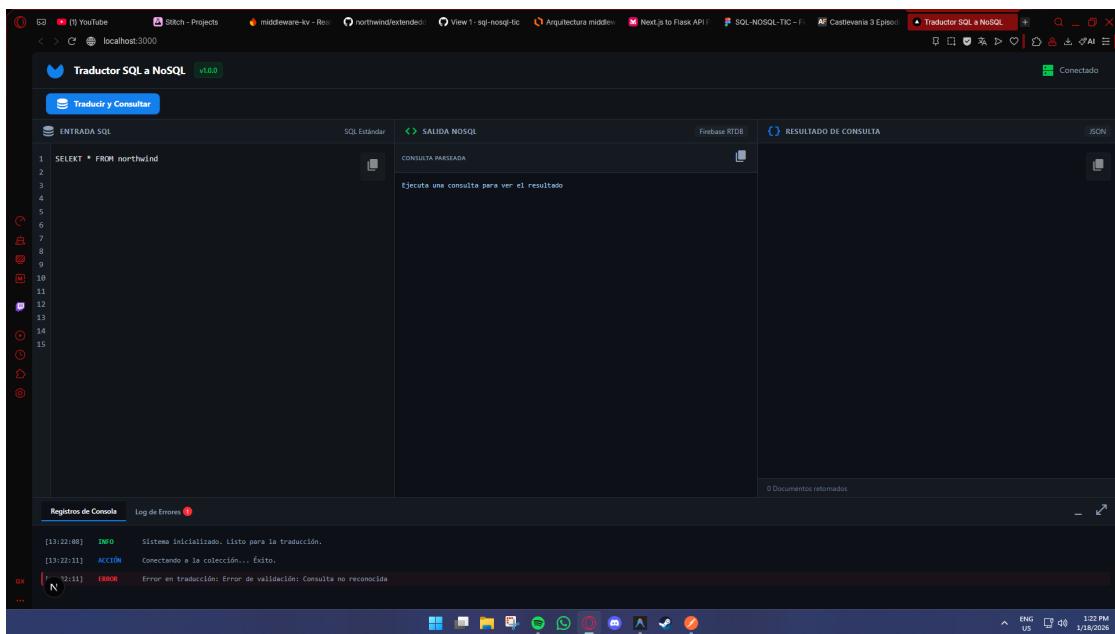


Figura 3.12: Error de sintaxis reportado en la interfaz

### Caso 13: Error semántico (Tabla inexistente)

**Objetivo:** Verificar la validación de existencia de colecciones.

Listing 3.13: Consulta SQL - Tabla No Encontrada

```
1 SELECT displayName , firstName FROM northwind2 WHERE lastName = 'Gaines';
```

**Resultado:** El adaptador de Firebase intentó acceder a la referencia `northwind2`, retornando un conjunto vacío. El sistema manejó esto correctamente indicando "0 documentos encontrados".

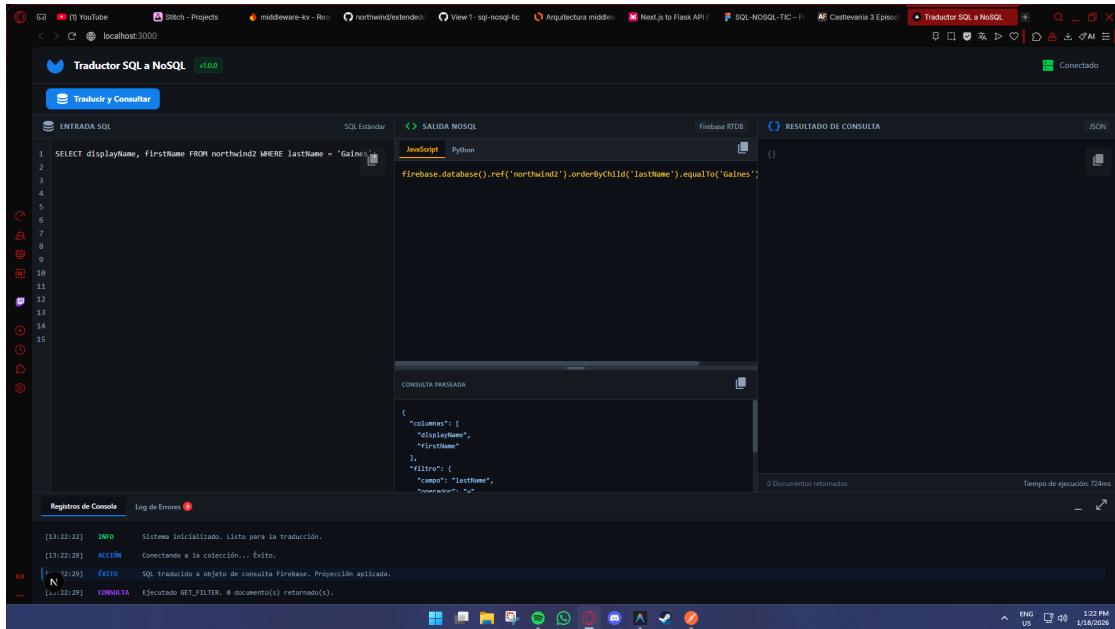


Figura 3.13: Manejo de consulta a colección inexistente

### 3.1.5. Pruebas no funcionales: Usabilidad

Se realizó una **evaluación heurística** completada aplicando los 10 principios de usabilidad de Jakob Nielsen. Esta metodología permitió inspeccionar la interfaz para identificar problemas de usabilidad y clasificarlos según su severidad en una escala del 0 (Sin problema) al 4 (Catástrofe de usabilidad).

#### Resultados de la evaluación heurística

A continuación se detalla el análisis punto por punto:

##### H1: Visibilidad del estado del sistema (Severidad: 0)

El sistema proporciona retroalimentación inmediata mediante el indicador de carga “Procesando...” en la barra de herramientas y mensajes de estado en la consola (“**ACCIÓN**”, “**ÉXITO**”), manteniendo al usuario informado.

##### H2: Correspondencia con el mundo real (Severidad: 0)

Se utiliza terminología estándar del dominio (SQL, Colección, JSON, Python), adecuada para el perfil técnico del usuario objetivo.

### H3: Control y libertad del usuario (Severidad: 0)

La interfaz permite redimensionar los paneles de salida y minimizar/maximizar la consola a demanda, otorgando control sobre el espacio de trabajo.

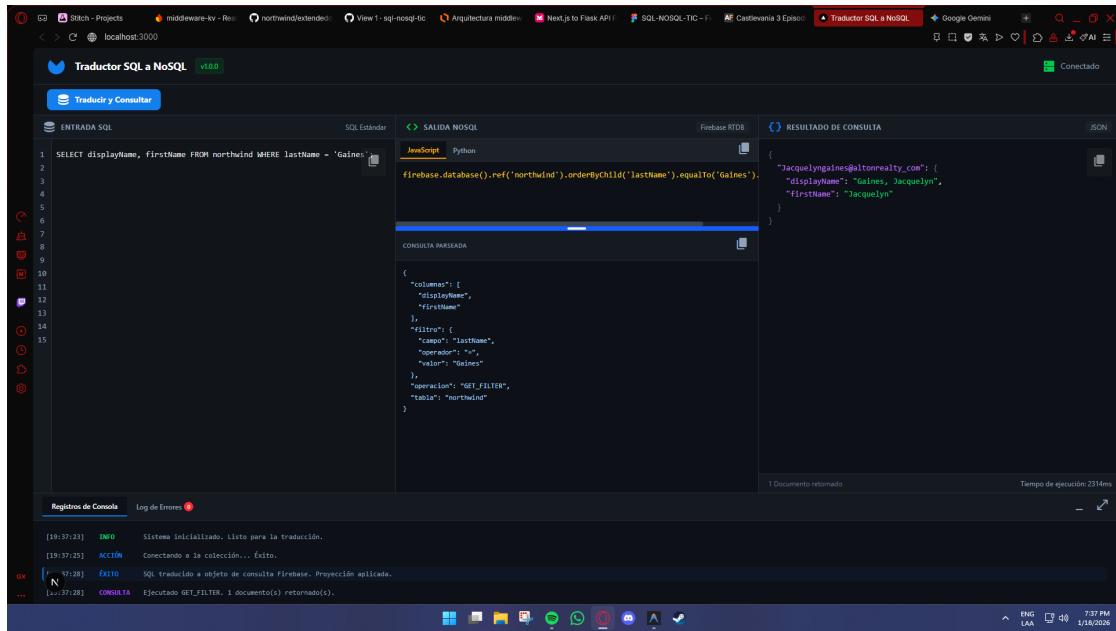


Figura 3.14: Usuario redimensionando el panel de salida (H3)

### H4: Consistencia y estándares (Severidad: 0)

Los colores de sintaxis (azul para keywords, verde para strings), iconos y tipografía se mantienen consistentes en toda la aplicación gracias al sistema de diseño.

### H5: Prevención de errores (Severidad: 0)

Se validaron mecanismos preventivos: el sistema bloquea el envío si el campo SQL está vacío o si no hay conexión con el backend, mostrando un mensaje claro antes de generar una excepción.

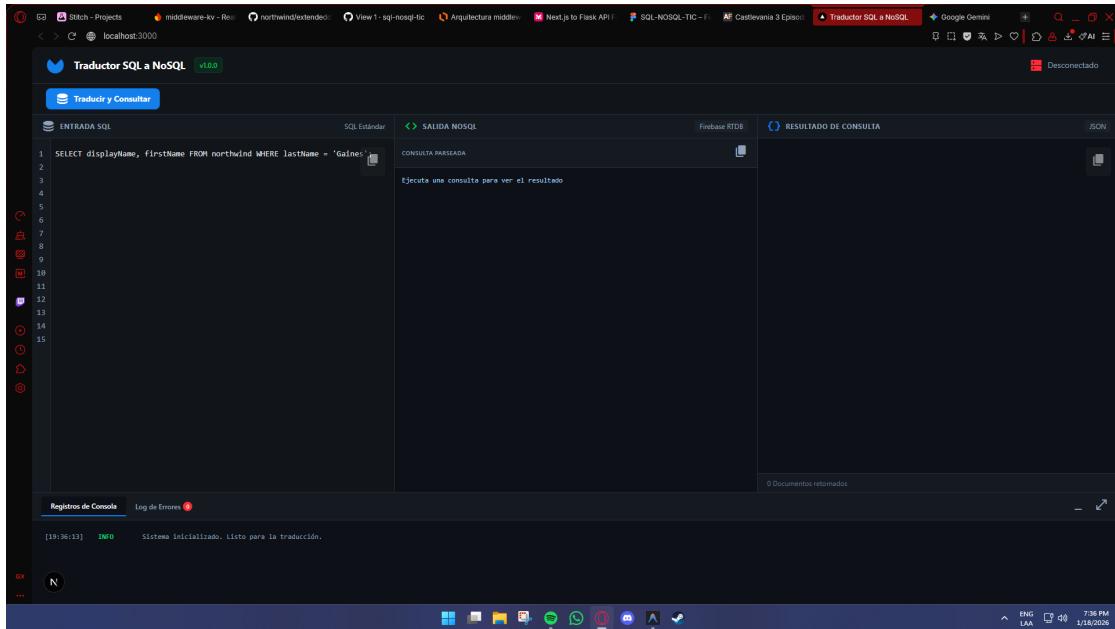


Figura 3.15: Bloqueo de ejecución por falta de conexión (H5)

## H6: Reconocimiento antes que recuerdo (Severidad: 0)

Los snippets generados (Python/JS) están visibles junto al resultado, evitando que el usuario deba recordar la sintaxis de Firebase SDK.

## H7: Flexibilidad y eficiencia de uso (Severidad: 1)

Aunque la consola permite filtros rápidos y expansión, la interfaz carece de atajos de teclado (ej. Ctrl+Enter para ejecutar), lo cual es una mejora deseable para usuarios expertos.

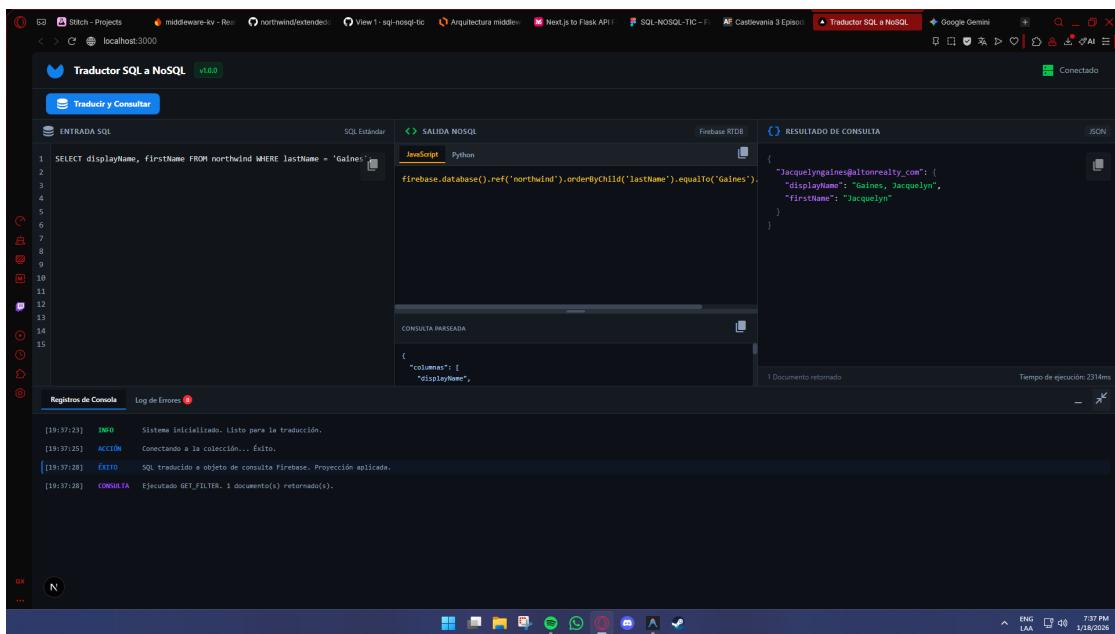


Figura 3.16: Consola expandida para revisión detallada (H7)

## H8: Estética y diseño minimalista (Severidad: 0)

El diseño en modo oscuro reduce la fatiga visual y elimina elementos distractores, centrándolo en el código.

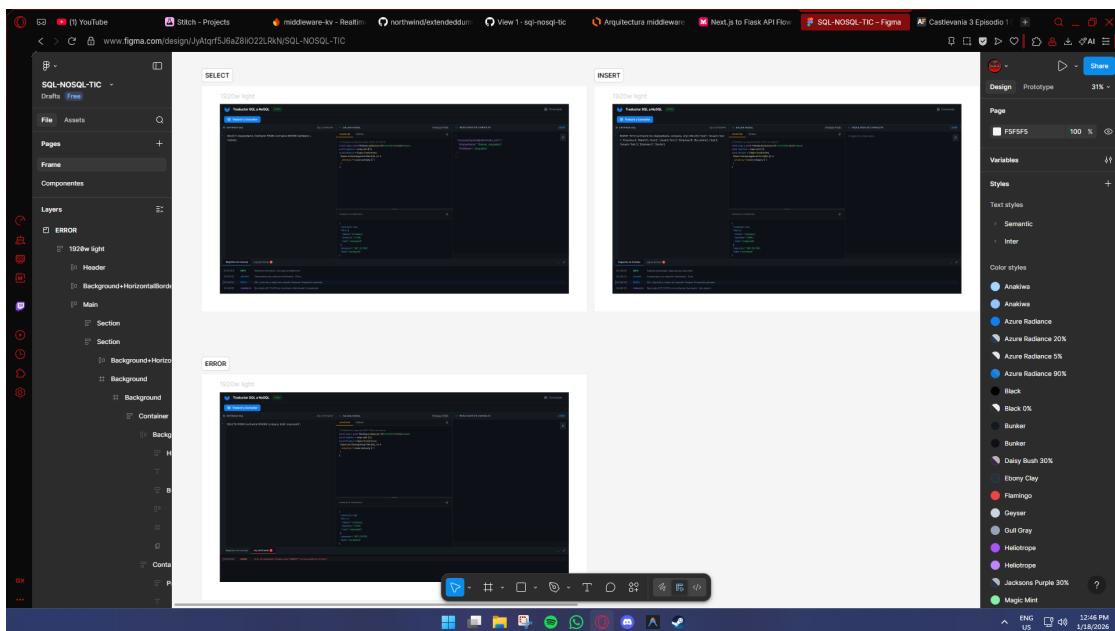


Figura 3.17: Interfaz minimalista en modo oscuro (H8)

## H9: Ayuda a reconocer y recuperar errores (Severidad: 0)

Los mensajes de error de sintaxis son descriptivos e indican el token problemático, facilitando la corrección inmediata.

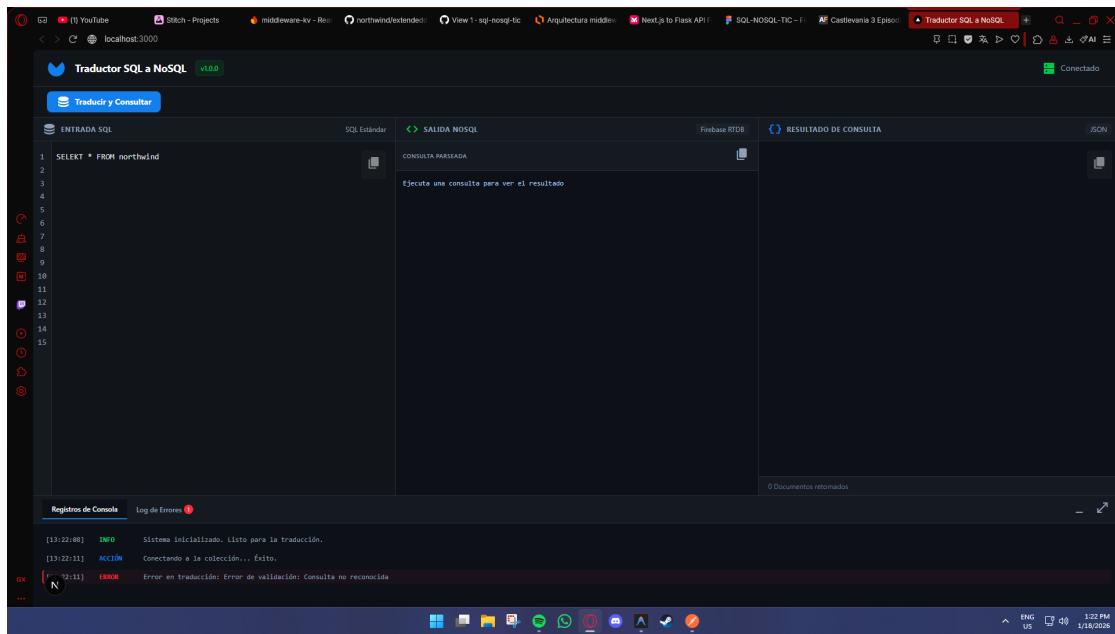


Figura 3.18: Mensaje de error descriptivo en consola (H9)

## H10: Ayuda y documentación (Severidad: 2)

El sistema carece de una sección de ayuda integrada o tooltips explicativos sobre las funcionalidades complejas, lo cual representa un problema de usabilidad menor.

### Reporte de severidad

La Tabla 3.1 resume los hallazgos. Se observa que la mayoría de los principios se cumplen satisfactoriamente (Severidad 0), existiendo oportunidades de mejora menores en flexibilidad y documentación.

Tabla 3.1: Matriz de Severidad de Usabilidad

Heurística	Observación	Severidad (0-4)
H1: Visibilidad	Indicadores de carga y logs claros.	0
H2: Mundo Real	Terminología técnica adecuada.	0
H3: Control	Paneles redimensionables y consola colapsable.	0
H4: Consistencia	Estilos visuales coherentes.	0
H5: Prevención	Validación de entrada y estado de red.	0
H6: Reconocimiento	Snippets de código visibles.	0
H7: Flexibilidad	Falta de atajos de teclado (Ctrl+Enter).	1
H8: Estética	Diseño limpio y modo oscuro.	0
H9: Recuperación	Errores de sintaxis descriptivos.	0
H10: Ayuda	Ausencia de documentación integrada.	2

### 3.2. Conclusiones

El desarrollo y validación del middleware SQL-NoSQL permite establecer las siguientes conclusiones con respecto a los objetivos planteados:

- **Viabilidad de la traducción SQL-NoSQL:** Se demostró que es posible traducir un subconjunto significativo del estándar SQL-92 (incluyendo proyecciones, filtros y patrones) a operaciones de Firebase Realtime Database. El uso de `sqlparse` combinado con expresiones regulares resultó ser una estrategia robusta para la normalización y análisis sintáctico de las sentencias, permitiendo una identificación precisa de la intención del usuario.
- **Eficacia de la estrategia híbrida:** La implementación de una estrategia de ejecución híbrida (Fetch & Filter) permitió superar las limitaciones nativas de Firebase (como la falta de soporte para `ILIKE` o condiciones `OR`). Si bien esta aproximación transfiere carga computacional al middleware, las pruebas funcionales confirmaron su eficacia para colecciones de tamaño moderado, ofreciendo una flexibilidad de consulta que la base de datos no posee por sí misma.
- **Calidad de la experiencia de usuario:** La Evaluación Heurística confirmó que la interfaz cumple satisfactoriamente con los estándares de la industria, obteniendo una

calificación de **Severidad 0** en principios críticos como prevención de errores y control del usuario. Si bien se detectaron oportunidades de mejora en la documentación integrada (Severidad 2), el sistema demostró ser intuitivo y seguro para el perfil técnico objetivo, validando el diseño minimalista centrado en el código.

- **Separación de responsabilidades:** La arquitectura por capas implementada, con un desacoplamiento claro entre el Parser, el Controlador y la Interfaz de Usuario, facilitó la evolución independiente del Frontend y Backend. Esto se evidenció durante el desarrollo, donde cambios en la lógica de 'store' del frontend (paso de Context a Hooks) no impactaron la estabilidad de la API de traducción.

### 3.3. Recomendaciones

A partir de los hallazgos y limitaciones identificadas durante el desarrollo, se proponen las siguientes recomendaciones para trabajos futuros:

- **Optimización mediante índices:** Para mitigar la latencia en la estrategia híbrida cuando el volumen de datos escala, se recomienda implementar un sistema que sugiera o cree índices automáticamente en Firebase (reglas `.indexOn`) basados en los campos más consultados en la cláusula WHERE.
- **Implementación de paginación en el servidor:** Actualmente, las consultas SELECT \* recuperan la colección completa. Se sugiere implementar paginación basada en cursor (`limitToFirst`, `startAt`) para manejar grandes volúmenes de datos sin saturar la memoria del middleware ni el ancho de banda del cliente.
- **Seguridad y autenticación:** Integrar Firebase Authentication para gestionar usuarios y roles. Esto permitiría no solo proteger los endpoints, sino también personalizar el acceso a los datos (Row Level Security) utilizando las Reglas de Seguridad de Firebase, traduciendo permisos SQL (GRANT/REVOKE) a reglas JSON.
- **Soporte multi-motor:** Abstraer la capa de adaptador de base de datos para soportar otros motores NoSQL populares como MongoDB o Amazon DynamoDB. La estructura actual del parser genera un AST agnóstico que podría transpilarse a otros lenguajes de consulta (ej. MQL de Mongo) con cambios mínimos en el núcleo.
- **Refinamiento de usabilidad:** Atendiendo a los hallazgos de la evaluación heurística, se sugiere incorporar una sección de "Ayuda Contextual"(Toolips) para funciones

avanzadas y habilitar atajos de teclado (ej. Ctrl+Enter para ejecutar), mejorando la eficiencia para usuarios expertos.

## Capítulo 4

# REFERENCIAS BIBLIOGRÁFICAS

- [1] B. Namdeo y U. Suman, «A Middleware Model for SQL to NoSQL Query Translation,» *Indian Journal of Science and Technology*, vol. 15, n.º 16, págs. 718-728, 2022. DOI: 10.17485/IJST/v15i16.2250
- [2] A. Dede y F. Ozcan, «When Relational-Based Applications Go to NoSQL Databases,» *2019 4th International Conference on Computer Science and Engineering (UBMK)*, págs. 1-6, 2019. DOI: 10.1109/UBMK.2019.8907131
- [3] J. S. Queiroz, T. A. Falcão, P. M. Furtado et al., «AMANDA: A Middleware for Automatic Migration between Different Database Paradigms,» *Applied Sciences*, vol. 12, n.º 12, pág. 6106, 2022. DOI: 10.3390/app12126106
- [4] R. Murthy, «Design and Development of a Query Processor for Heterogeneous NoSQL Databases,» Tesis doct., Sathyabama Institute of Science y Technology, 2021. dirección: <http://hdl.handle.net/10603/376288>
- [5] M. N. Mami, D. Graux, S. Scerri et al., «The Query Translation Landscape: A Survey,» *arXiv preprint arXiv:1910.03118*, 2019.
- [6] J. Melton y A. R. Simon, *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.
- [7] K. Delaney, *Microsoft SQL Server 2012 Internals*. Microsoft Press, 2013.
- [8] A. Corbellini, C. Mateos, A. Zunino et al., «Persisting big-data: The NoSQL landscape,» *Information Systems*, vol. 63, págs. 1-23, 2016. DOI: 10.1016/j.is.2016.07.009
- [9] G. DeCandia et al., «Dynamo: Amazon's highly available key-value store,» en *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ACM, 2007, págs. 205-220.

- [10] Google Developers, *Firebase Documentation*, 2024. dirección: <https://firebase.google.com/docs>
- [11] TIOBE Index, *TIOBE Programming Community Index for 2024*, 2024. dirección: <https://www.tiobe.com/tiobe-index/>
- [12] N. Gift, K. Behrman, A. Deza y G. Ghemawat, *Python for DevOps: Learn Ruthlessly Effective Automation*. O'Reilly Media, 2020.
- [13] A. Albrecht, *sqlparse: A non-validating SQL parser for Python*, <https://github.com/andialbrecht/sqlparse>, 2023.
- [14] M. Lutz, *Programming Python*, 4th. O'Reilly Media, 2011.
- [15] Microsoft, *Visual Studio Code Documentation*, 2023. dirección: <https://code.visualstudio.com/docs>
- [16] S. Chacon y B. Straub, *Pro Git*, 2nd. Apress, 2014. dirección: <https://git-scm.com/book/en/v2>
- [17] GitHub Inc., *GitHub Docs: Collaborating with issues and pull requests*, 2023. dirección: <https://docs.github.com>
- [18] Figma Inc., *Figma: Collaborative Interface Design Tool*, 2023. dirección: <https://www.figma.com>
- [19] K. Relan, *Building REST APIs with Flask: Create Python Web Services with MySQL*. Apress, 2019. DOI: 10.1007/978-1-4842-5021-1
- [20] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2.<sup>a</sup> ed. O'Reilly Media, 2018.
- [21] A. Banks y E. Porcello, *Learning React: Modern Patterns for Developing React Apps*, 2.<sup>a</sup> ed. O'Reilly Media, 2020, ISBN: 9781492051725.
- [22] S. Hoque, *Full-Stack React Projects: Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node.js*, 2.<sup>a</sup> ed. Packt Publishing, 2020, ISBN: 9781839215414.
- [23] Vercel, *Next.js Documentation*, 2024. dirección: <https://nextjs.org/docs>
- [24] K. Schwaber y J. Sutherland, *The Scrum Guide*. Scrum.org, 2020.
- [25] J. Sutherland y K. Schwaber, *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*. Scrum.org, 2014.
- [26] I. Sommerville, *Software Engineering*, 9th. Addison-Wesley, 2011.

- [27] I. Burnstein, *Practical Software Testing: A Process-Oriented Approach*. Springer, 2003.
- [28] D. M. Beazley, *PLY (Python Lex-Yacc)*, 2023. dirección: <http://www.dabeaz.com/ply/>
- [29] Meta Platforms, Inc., *React – A JavaScript library for building user interfaces*, 2024. dirección: <https://react.dev>
- [30] Vercel Inc., *Data Fetching and Rendering in Next.js*, 2024. dirección: <https://nextjs.org/docs>
- [31] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.
- [32] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th. McGraw-Hill, 2009.
- [33] K. Beck et al., *Manifesto for Agile Software Development*, <https://agilemanifesto.org>, 2001.

## ANEXO I

# Conjunto de datos (Dataset NoSQL)

A continuación se presenta un extracto representativo de la estructura de datos migrada a Firebase Realtime Database. Este formato muestra cómo las entidades relacionales (en este caso, Empleados) fueron transformadas a un modelo orientado a documentos JSON, utilizando claves compuestas (email) para optimizar la búsqueda.

Listing I.1: Ejemplo de estructura JSON en Firebase (Colección Empleados)

```
1  {
2    "northwind": {
3      "iconley@altonrealty_com": {
4        "displayName": "Aileen , Isabelle",
5        "company": "Alton Realty",
6        "title": "Accountant",
7        "firstName": "Isabelle",
8        "lastName": "Aileen",
9        "email": "iconley@altonrealty.com",
10       "commonId": 63311,
11       "team": "Finance",
12       "subTeam": 0,
13       "hireDate": "3/17/2007",
14       "fax": "(313) 068-3076",
15       "state": "Michigan",
16       "zip": 48207,
17       "officeCity": "Detroit",
18       "teamLeader": "Katherine Castillo",
19       "office": "The Palace",
20       "pictureUrl": 0,
21       "extension": 83076,
22       "skills": [
```

```
        "Health",
        "Developers",
        "Craft",
        "Drama",
        "Drafters"
    ],
    "interests": [
        "Tai chi",
        "Beach Volleyball",
        "Kayaking",
        "Basketball",
        "Association football"
    ]
},
"jbird@altonrealty_com": {
    "displayName": "Bird, Joy",
    "title": "Accountant",
    "email": "jbird@altonrealty.com",
    "commonId": 63936,
    "team": "Finance",
    "hireDate": "9/15/2010",
    "skills": [
        "Philosophy",
        "Political",
        "Coaches"
    ]
}
}
}
```

## **ANEXO II**

# **Detalle de planificación (Backlog)**

Este anexo detalla las tareas técnicas, tipos de trabajo y estimaciones horarias para cada Historia de Usuario implementada, complementando la visión general presentada en el Capítulo 2.

Tabla II.1: Backlog HU01 — Ingresar una consulta SQL

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T1.1	Diseñar la sección de entrada de texto para consultas SQL.	Diseño UI	4
T1.2	Implementar campo de texto con resaltado de sintaxis.	Desarrollo frontend	6
T1.3	Validar la estructura básica de sentencias SQL (SELECT, FROM, WHERE).	Desarrollo backend	6
T1.4	Probar la entrada y validación con ejemplos de consultas simples.	Pruebas funcionales	4
T1.5	Revisar la legibilidad y facilidad de uso en distintos tamaños de pantalla.	Validación UX	2

Tabla II.2: Backlog HU02 — Ejecutar la consulta SQL

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T2.1	Crear botón de ejecución y su evento asociado.	Desarrollo frontend	4
T2.2	Configurar endpoint Flask para recibir la consulta.	Desarrollo backend	6
T2.3	Integrar comunicación HTTP Frontend → Backend.	Integración	6
T2.4	Manejar errores de ejecución (sintaxis, conexión, timeout).	Backend	4
T2.5	Implementar mensajes de resultado y notificaciones visuales.	Frontend	4
T2.6	Probar flujo completo con consultas válidas e inválidas.	Pruebas	4

Tabla II.3: Backlog HU03 — Ver la traducción generada de la consulta

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T3.1	Diseñar panel para mostrar traducción NoSQL.	Diseño UI	4
T3.2	Implementar módulo de traducción SQL → NoSQL (clave–valor).	Desarrollo backend	10
T3.3	Mostrar la traducción generada en tiempo real en el panel.	Frontend	6
T3.4	Agregar opción para copiar la traducción al portapapeles.	Frontend	2
T3.5	Validar precisión de la traducción con ejemplos.	Pruebas	4

Tabla II.4: Backlog HU04 — Identificar errores en la traducción

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T4.1	Definir estructura de mensajes de error de traducción.	Backend	4
T4.2	Implementar manejo de errores detallado en módulo traductor.	Backend	6
T4.3	Mostrar mensajes en interfaz junto a la consulta.	Frontend	4
T4.4	Validar distintos escenarios de error (palabras clave, sintaxis).	Pruebas	4
T4.5	Revisar usabilidad y comprensión de los mensajes.	UX	2

Tabla II.5: Backlog HU05 — Visualizar resultados de la ejecución

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T5.1	Diseñar componente de tabla para mostrar resultados.	Diseño UI	4
T5.2	Implementar renderizado dinámico de datos en tabla.	Frontend	6
T5.3	Recibir respuesta desde el backend (JSON).	Integración	4
T5.4	Mostrar cantidad de registros y tiempo de ejecución.	Frontend	4
T5.5	Probar visualización con distintos tamaños de resultados.	Pruebas	4

Tabla II.6: Backlog HU06 — Ver consulta original y traducción simultáneamente

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T6.1	Diseñar disposición doble de paneles (SQL / NoSQL).	Diseño UI	4
T6.2	Implementar vista con sincronización de desplazamiento.	Frontend	6
T6.3	Agregar control para ocultar/mostrar panel de traducción.	Frontend	4
T6.4	Validar que ambas vistas se actualicen tras ejecutar consulta.	Pruebas	4
T6.5	Revisar la alineación visual y consistencia de formato.	UX	2

Tabla II.7: Backlog HU07 — Interfaz intuitiva y ordenada

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T7.1	Unificar esquema visual (tipografía, colores, espacio).	Diseño UI	4
T7.2	Revisar jerarquía visual y consistencia entre secciones.	UX	4
T7.3	Implementar mejoras visuales (íconos, bordes, layout).	Frontend	6
T7.4	Validar la experiencia de usuario con feedback de compañeros.	Validación	2
T7.5	Documentar lineamientos visuales básicos.	Documentación	2

Tabla II.8: Backlog HU08 — Retroalimentación visual durante la ejecución

ID Tarea	Descripción de la tarea	Tipo	Estimación (h)
T8.1	Diseñar animación o indicador de progreso.	Diseño UI	3
T8.2	Implementar indicador de carga en frontend.	Frontend	4
T8.3	Integrar estados de procesamiento, éxito y error.	Frontend	4
T8.4	Validar que el indicador desaparezca correctamente al finalizar.	Pruebas	3
T8.5	Probar con ejecuciones largas o con error simulado.	QA	2

Tabla II.9: Totales aproximados por categoría

Categoría	Horas estimadas
Diseño UI/UX	39
Desarrollo frontend	48
Desarrollo backend	32
Integración	10
Pruebas / QA	23
Documentación / Validación	8
<b>Total general</b>	$\approx 160$ horas técnicas

## ANEXO III

# Guía de instalación y despliegue

En este anexo se detallan los pasos necesarios para desplegar la aplicación en un entorno local, asegurando la reproducibilidad de los resultados presentados en esta tesis.

## Requisitos previos

- **Python:** Versión 3.8 o superior.
- **Node.js:** Versión 18.x o superior.
- **Firebase:** Proyecto activo en Firebase Console con Realtime Database habilitado.

## Pasos de instalación

### 1. Clonar el repositorio

```
1 git clone https://github.com/greyox97/sql-nosql-tic.git  
cd sql-nosql-tic
```

### 2. Configurar el Backend (Flask)

Navegar al directorio raíz del backend y crear un entorno virtual:

```
1 cd backend  
python -m venv venv
```

```
source venv/bin/activate # En Windows: venv\Scripts\activate  
pip install -r requirements.txt
```

Configurar las variables de entorno creando un archivo `.env` en la raíz del backend:

```
1 FIREBASE_CREDENTIALS=ruta/a/tus/credenciales.json  
FIREBASE_DB_URL=https://tu-proyecto.firebaseio.com/
```

### 3. Configurar el Frontend (Next.js)

Navegar al directorio del frontend e instalar dependencias:

```
1 cd ../frontend  
npm install
```

### 4. Ejecución

Iniciar los servidores de desarrollo en dos terminales separadas:

**Terminal 1 (Backend):**

```
1 flask run --debug --port 5000
```

**Terminal 2 (Frontend):**

```
1 npm run dev
```

La aplicación estará disponible en `http://localhost:3000`.

## ANEXO IV

# Enlaces y recursos

En esta sección se listan los recursos digitales, repositorios y herramientas utilizados durante el desarrollo de la investigación.

Tabla IV.1: Enlaces a repositorios y recursos

Recurso	Enlace URL
Repositorio GitHub del proyecto	<a href="https://github.com/greyox97/sql-nosql-tic">https://github.com/greyox97/sql-nosql-tic</a>
Dataset original (Northwind)	<a href="https://github.com/haqueabida/northwind">https://github.com/haqueabida/northwind</a>
Prototipos UI (Figma)	<a href="https://www.figma.com/design/JyAtqrf5J6aZ8Ii022LRkN/SQL-NOSQL-TIC">https://www.figma.com/design/JyAtqrf5J6aZ8Ii022LRkN/SQL-NOSQL-TIC</a>
Diagramas de arquitectura (Mermaid)	<a href="https://mermaid.ai/app/projects/c4832a57-203f-431a-8c93-bcf2cd8c673e">https://mermaid.ai/app/projects/c4832a57-203f-431a-8c93-bcf2cd8c673e</a>
Diagramas de flujo (Lucidchart)	<a href="https://lucid.app/lucidchart/b195567a-b30e-42c9-8504-e67454d77688">https://lucid.app/lucidchart/b195567a-b30e-42c9-8504-e67454d77688</a>

## Herramientas y librerías principales

- **sqlparse (Python):** <https://github.com/andialbrecht/sqlparse>
- **Firebase Admin SDK:** <https://firebase.google.com/docs/admin/setup>
- **Next.js Framework:** <https://nextjs.org/>